Symmetric chain decompositions of partially ordered sets

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ondrej Zjevik

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

John Greene, Dalibor Froncek

July, 2014

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school. First, I have to express my deep appreciation to prof. John Greene who has been providing me with outstanding guidelines and recommendations ever since he introduced me to this topic. I am especially grateful for his persistence in reading various copies of this paper.

I am indebted to prof. Dalibor Froncek for encouraging me to apply to UMD, his support and for serving on my committee.

I would also like to thank prof. Dougles Dunham for serving on my committee.

Abstract

A partially ordered set, or poset, is a set of elements and a binary relation which determines an order within elements. Various combinatorial properties of finite and ordered posets have been extensively studied during the last 4 decades. The Sperner property states that the size of the largest subset of pairwise incomparable elements does not exceed the size of the largest level set in an ordered poset. Since a symmetric chain decomposition is a sufficient condition for the Sperner property, we may prove the Sperner property by finding a symmetric chain decomposition for a poset.

In this paper we focus on three types of posets: the Boolean algebra, inversion poset and the Young's lattice. An explicit construction for a symmetric chain decomposition is known only for Boolean algebras. No explicit construction has been found for inversion posets and Young's lattices, a symmetric chain decomposition was found only for a small subset of these posets. Using a maximal flow, we introduce an algorithm for finding this decomposition. We present our results and discuss two implementations of this algorithm.

Contents

List of Tables

List of Figures

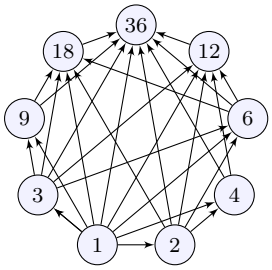
Chapter 1

Introduction

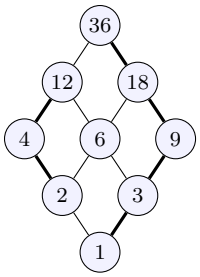
A partially ordered set (poset) is an ordered pair of a set with an order relation, e.g., $(\mathcal{P}, \mathcal{R})$. This relation $\mathcal{R}$ is

- *reflexive*: each element is related to itself, i.e., $\forall a \in \mathcal{P}, (a, a) \in \mathcal{R}$,
- *transitive*: if $(a, b) \in \mathcal{R}$ and $(b, c) \in \mathcal{R}$, then $(a, c) \in \mathcal{R}$,
- *antisymmetric*: if $(a, b) \in \mathcal{R}$ and $a \neq b$, then $(b, a) \notin \mathcal{R}$.

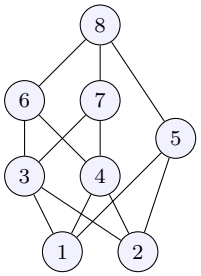
If two elements are related, $(a, b) \in \mathcal{R}$, we use $a \leq b$ to denote this relation. If we would like to emphasis a specific poset $\mathcal{P}$ we use $a \leq_P b$ instead. The set $\mathcal{P}$ with the relation $\mathcal{R}$ can be considered as a directed graph where $\mathcal{P}$ is the vertex set and $\mathcal{R}$ is the edge set. The vertex set $\mathcal{P}$ can be finite or infinite. A simple example is when $\mathcal{P} = \mathbb{Z}^+$ and the relation is defined as $a \leq b$ if $b - a \in \mathbb{N}$. This poset is infinite and since we can compare each pair of positive integers, one integer is always less or equal then another, we say that this order is *total*. A *partial* order on the same set $\mathcal{P} = \mathbb{Z}^+$ is given by divisibility. That is, $a \leq b$ when b is divisible by a. For this order relation, not every pair of positive numbers is comparable. For example if we consider 3 and 5, neither 3 divides 5 nor 3 is divisible by 5. We say 3 and 5 are incomparable. If we restrict $\mathcal{P}$ to all divisors of 36 we get a finite poset. These types of posets are called *division posets*, since the relation is



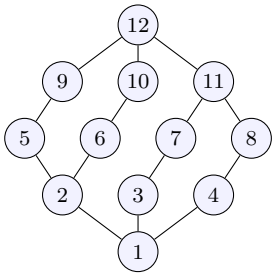
(a) Directed graph of a division poset D_{36}.



(b) Hasse diagram of a division poset D_{36} with a SSCD.



(c) Hasse diagram of a poset without a rank function.



(d) Hasse diagram of a ranked poset.

Figure 1.1: Examples of posets.

defined by division. Finite posets will be our main focus throughout this paper. More types of posets and precise definitions will be introduced in Chapter 2.

The directed graph of the finite division poset of divisors of 36 is shown in Figure 1.1(a). This is the only time we show the entire directed graph of a poset. We will use Hasse diagrams for graphical representations of posets because they contain the same amount of information but have fewer edges than directed graphs. An edge (a, b) is in a Hasse diagram if and only if there is no c such that oriented paths from a to c and from c to b would exist. Additionally, Hasse diagrams do not have oriented edges but we assume that every edge is oriented in upward direction. We can see the difference between a directed graph and a Hasse diagram of a division poset in Figures 1.1(a) and

1.1(b).

Posets have a variety of possible properties. A poset is *ranked* if elements of the Hasse diagram can be partitioned into horizontal level sets such that edges are only between the closest sets. Since there are no edges within each layer nor between layers which are separated by at least one other layer, every oriented path between two elements must have the same length. Each Hasse diagram in Figure 1.1 is ranked except (c). This poset is not ranked because there are two paths from 2 to 8 with different lengths. The path $2 \to 5 \to 8$ has length 2 but the path $2 \to 4 \to 7 \to 8$ has length 3.

An *upper bound* of a subset A of $\mathcal{P}$ is an element p such that $a \leq p$ for all $a \in A$. Let U be a set of all upper bounds of A. If there exists an upper bound q such that $q \leq p$ for all $p \in U$, we call q the *least upper bound* of A. *Lower bounds* and the *greatest lower bound* are defined similarly. Notice that the definition of the least upper bound and the greatest lower bound imply uniqueness, since the relation is antisymmetric.

A poset is called a *lattice* if the least upper bound and the greatest lower bound exist for every pair of elements. As we could suppose the division poset and the poset in Figure 1.1(d) are lattices, but the poset in Figure 1.1(c) is not a lattice. If we look at the subset $\{3, 4, 6, 7\}$ in Figure 1.1(c), we can find that $\{3, 4\}$ does not have the least upper bound and $\{6, 7\}$ does not have the greatest lower bound.

A *decomposition* of a poset is a partition of $\mathcal{P}$ into disjoint subsets $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k$ such that $\cup_{i=0}^{k} \mathcal{P}_i = \mathcal{P}$. There are many types of decomposition but we will focus on symmetric saturated chain decompositions (SSCD). An ordered n-tuple $(c_1, c_2, \ldots, c_n)$ is called a *chain*, with length $n-1$, of the poset $\mathcal{P}$ if $c_i \neq c_{i+1}$ and $c_i \leq c_{i+1}$ for $1 \leq i < n$. A chain $(c_1, c_2, \ldots, c_n)$ is *saturated* if it cannot be internally extended, i.e.; (c_i, c_{i+1}) is an edge in the Hasse diagram of the poset for $1 \leq i < n$. A saturated chain is *symmetric* if it starts and ends at levels whose distance from the middle level(s) of the poset is the same. Figure 1.1(b) contains a diagram with two highlighted symmetric saturated chains. A poset has a SSCD if $\mathcal{P}$ can be decomposed into symmetric saturated chains.

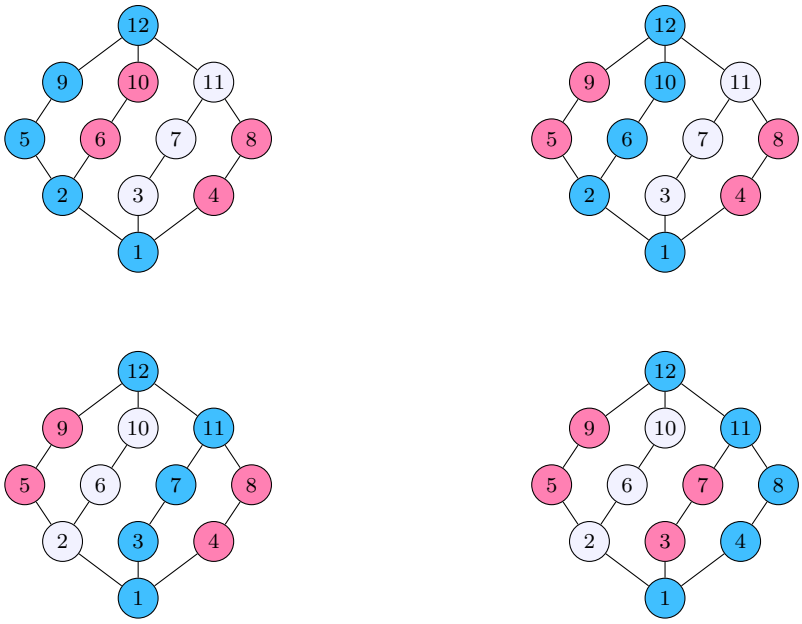


Figure 1.2: Different saturated chain decompositions of a poset depicted in figure 1.1(d).

The division poset (lattice) in Figure 1.1(b) is decomposable into three symmetric saturated chains. The chains, as highlighted in the Figure, are $(1, 3, 9, 18, 36)$, $(2, 4, 12)$ and (6). The lattice in Figure 1.1(d) cannot be decomposed into symmetric saturated chains. If this lattice were decomposable, there would have to be a longest chain from 1 to 12. We have exactly 4 choices for this chain, the choice depends on which element from the middle layer is included. Each choice will leave two non symmetric chains as we can see in Figure 1.2.

Before we introduce an algorithm for finding a symmetric saturated chain decomposition for a poset; fundamental terminology, a few properties of posets and essential theorems will be introduced in chapter 2. We also explain a connection between Sperner's property and the existence of a SSCD in this chapter.

The rest of the proofs are placed in chapter 3. In this chapter we describe isomorphisms between division and boolean posets and within inversion posets.

Chapter 4 introduces an algorithm for finding a SSCD for posets. This algorithm uses a maximum flow in a graph to construct a SSCD. Two different algorithms for finding a maximum flow in a graph are discussed.

Chapter 5 contains a brief description of two implementations of the algorithm from chapter 4. We provide a table with all posets for which our implementation finds a SSCD and discuss suggestions for a future work.

Both of our implementations have similar interface and a description of these interfaces is given in appendix A. A performance comparison between the two implementations, examples of a SSCD for selected posets and source codes of our implementations are placed in appendixes, also.

Chapter 2

Formal definitions and fundamental properties of posets

The theory of partially ordered sets investigates a poset as a graph or as a set of elements. This set can be infinite but we will focus on finite sets only. At the beginning of this chapter the most common notation and definitions from Graph Theory are introduced. Concrete examples of posets with known attributes will be introduced after definitions. Most of these definitions can be found in [1] or in [10].

Let $\mathcal{P}$ be a poset, $\mathcal{R}$ its relation and p, p', q, q' elements of $\mathcal{P}$.

We say that p and q are *comparable* if $(p, q) \in \mathcal{R}$ or $(q, p) \in \mathcal{R}$, and we typically write this as $p \leq q$ or $q \leq p$. In the case when p and q are comparable and $p \neq q$ either $p < q$ or $q < p$ is valid. An element p *covers* q, denoted as $q \lessdot p$, if $q < p$ and if $q < p' \leq p$ implies $p = p'$. An element p is a *minimal element* of $\mathcal{P}$ if $q \leq p$ implies $q = p$; a *maximal element* is defined similarly.

A ranked poset is equipped with a *rank function* ρ. This is a function from $\mathcal{P}$ to $\mathbb{N}$ satisfying $\rho(p) = \rho(q) + 1$ if p covers q. The rank of a poset is defined as the largest rank over all elements from the poset. Since every finite lattice has only one minimal element, the rank of this minimal element is zero.

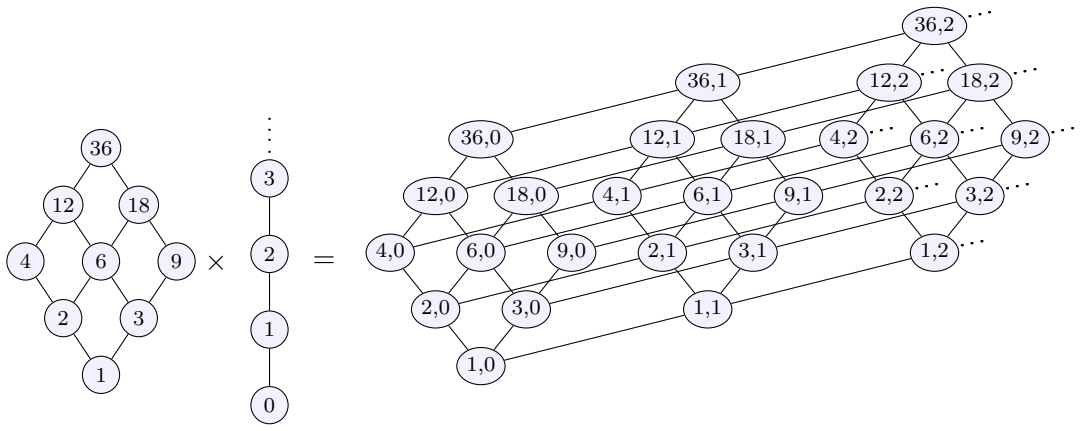


Figure 2.1: Example of a direct product of two ranked posets.

A chain $C = (c_1, c_2, \ldots, c_k)$ is *saturated* if $c_1 \lessdot c_2 \lessdot \cdots \lessdot c_k$. This chain is *maximal* if there is no c_0 such that $(c_0, c_1, c_2, \ldots, c_k)$ or $(c_1, c_2, \ldots, c_k, c_0)$ would be a valid chain in $\mathcal{P}$. Each of the highlighted chains in Figure 1.2 is saturated.

A saturated chain $(c_1, c_2, \ldots, c_n)$ is symmetric if $\rho(c_1) + \rho(c_n) = \rho(\mathcal{P})$. Note that $\rho(c_1) + \rho(c_n) = \rho(\mathcal{P})$ implies that $\rho(c_{1+i}) + \rho(c_{n-i}) = \rho(\mathcal{P})$ for $0 \leq i < n/2$.

A set $\mathcal{Q} \subset \mathcal{P}$ is an *antichain* if every pair of elements in $\mathcal{Q}$ is incomparable, i.e., there are no $p, q \in \mathcal{Q}$ such that $p \neq q$ and $p \leq q$ or $q \leq p$. Sets $\{2, 7, 8\}, \{2, 3, 4\}, \{6, 9, 11\}$, $\{5, 6, 7, 8\}$ are examples of antichains in Figure 1.2. The *direct product* of two posets $(\mathcal{P}, \mathcal{R})$ and $(\mathcal{Q}, \mathcal{S})$, $(\mathcal{P}, \mathcal{R}) \times (\mathcal{Q}, \mathcal{S})$, is a poset with the set defined as a Cartesian product of $\mathcal{P}$ and $\mathcal{Q}$, the order relation of $\mathcal{P} \times \mathcal{Q}$ is given by $(p, q) \leq (p', q')$ if and only if $p \leq_{\mathcal{P}} p'$ and $q \leq_{\mathcal{Q}} q'$ in $\mathcal{R}$ and $\mathcal{S}$, respectively. Consider a direct product of divisors of 36 ordered by divisibility and $\mathbb{N}$ ordered by a total order $p \leq q$ if $q - p \in \mathbb{N}$ shown in Figure 2.1. An example of a saturated chain is $\Big((1,0), (2,0), (2,1), (6,1), (6,2)\Big)$. Since 2 and 3 are incomparable in the division poset, the set $\{(2,0), (3,1)\}$ is an antichain in $\mathcal{P} \times \mathcal{Q}$. If $\mathcal{P}$ and $\mathcal{Q}$ are ranked, then so is $\mathcal{P} \times \mathcal{Q}$ by $\rho((p, q)) = \rho_{\mathcal{P}}(p) + \rho_{\mathcal{Q}}(q)$

If $\mathcal{P}$ is a ranked poset, its elements can be partitioned into layers, the ith layer of a ranked poset $\mathcal{P}$ is denoted as $N_i(\mathcal{P}) := \{p \in \mathcal{P} | \rho(p) = i\}$. The size of $N_i(\mathcal{P})$ is called the ith *Whitney* number, denoted by $W_i = |N_i(\mathcal{P})|$. Note that $W_j = 0$ if $j < 0$ or $j > \rho(\mathcal{P})$

from the definition of $N_j(\mathcal{P})$, since each layer $N_j(\mathcal{P})$ is empty. A *generating* function of a ranked poset $\mathcal{P}$ is given by

$$GF(\mathcal{P}) = \sum_{w \in \mathcal{P}} q^{\rho(w)} = \sum_{i=0}^{\rho(\mathcal{P})} W_i q^i.$$

We stated that posets in Figures 1.1(b) and 1.1(d) are ranked. Hence, there is a generating function for each poset. The generating function for the poset in Figure 1.1(d) is $1 + 3q + 4q^2 + 3q^3 + q^4$, $GF(D_{36}) = 1 + 2q + 3q^2 + 2q^3 + q^4$.

A poset is *rank unimodal* if there exist j such that $W_0 \leq W_1 \leq \cdots \leq W_{j-1} \leq W_j \geq W_{j+1} \geq \cdots \geq W_{\rho(\mathcal{P})}$. A *rank symmetric* poset is a ranked poset whose Whitney numbers are symmetric with respect to the middle layer(s), i.e.; $W_i = W_{\rho(\mathcal{P})-i}$ for $0 \leq i < \rho(\mathcal{P})/2$. Again, if we look at Figures 1.1(b) and 1.1(d), sequences of Whitney numbers are $(1, 2, 3, 2, 1)$ and $(1, 3, 4, 3, 1)$, respectively. Both sequences are symmetric and unimodal but only the division poset has a SSCD.

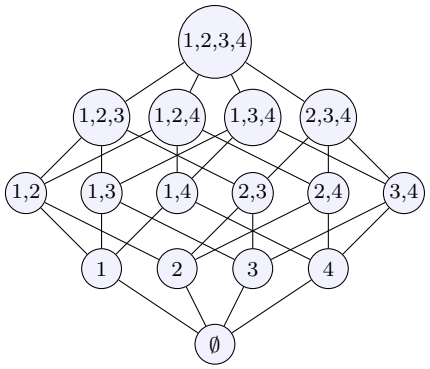
Posets $\mathcal{P}$ and $\mathcal{Q}$ are *isomorphic*, $\mathcal{P} \cong \mathcal{Q}$, if there is a bijective mapping φ from $\mathcal{P}$ onto $\mathcal{Q}$ such that $p \leq_{\mathcal{P}} q$ if and only if $\varphi(p) \leq_{\mathcal{Q}} \varphi(q)$.

Theorem 2.1 (Sperner's Theorem)**.** *Let n be a positive integer and $\mathcal{F}$ be a family of subsets of $[n] := \{1, 2, \ldots, n\}$ such that no member of $\mathcal{F}$ is included in another member of $\mathcal{F}$, that is, for all $X, Y \in \mathcal{F}$ we have $X \not\subset Y$. Then*

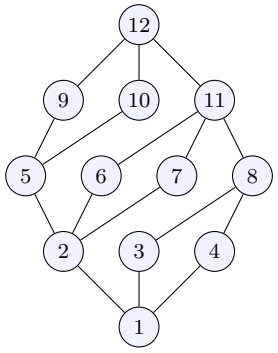
$$|\mathcal{F}| \leq \begin{cases} \binom{n}{n/2} & \text{if } n \text{ is even,} \\ \binom{n}{(n-1)/2} & \text{if } n \text{ is odd.} \end{cases}$$

Sperner's Theorem was at the beginning of the Poset Theory and many important results were obtained by using this theorem. Three different proofs of this theorem can be found in [4], as well as other important results. We will prove this theorem in the next chapter using a symmetric chain decomposition of the family of subsets of $\{1, 2, \ldots, n\}$.

Since each level set of a poset contains only incomparable elements, a lower bound



(a) B_4



(b) A lattice without the Sperner property.

Figure 2.2: Examples of two different lattices.

for the size of the largest antichain is given by the largest Whitney number. Sperner's Theorem was stated for the family of subsets of a set ordered by inclusion and it says that this lower bound is tight, the size of the biggest antichain is equal to the largest Whitney number. A generalization of this theorem is available for other types of posets; a ranked poset $\mathcal{P}$ has the *Sperner property* if the size of any antichain is less than or equal to the biggest Whitney number.

The two lattices in Figure 1.1 both have the Sperner property. However, the lattice in Figure 2.2(b) is rank symmetric but it does not have the Sperner property nor a SSCD, the antichain $\{3, 4, 5, 6, 7\}$ is larger than the size of the middle layer.

Unfortunately, it is not true that every poset with the Sperner property has a symmetric saturated chain decomposition. An example of such a poset is in Figure 1.1(d). This poset satisfies the Sperner property, the size of each antichain is not bigger than 4 which is the size of the biggest antichain formed by the middle layer. However, there is no SSCD of this poset as we discussed in Chapter 1.

We will find a use of *q-functions* in examining rank functions. A *q-analog* of n is defined as $\{n\}_q = \frac{1-q^n}{1-q}$ and we often want to evaluate the q-analog of n for $q = 1$. This

value does not exist, but we can find a limit as q approaches 1,

$$\lim_{q\to 1}\frac{1-q^n}{1-q}=\lim_{q\to 1}\frac{(1-q)(1+q+q^2+\cdots+q^{n-1})}{1-q}=\lim_{q\to 1}1+q+q^2+\cdots+q^{n-1}=n.$$

Additionally, we will use a continuous extension of q-analog of n,

$$\frac{1-q^n}{1-q}\equiv 1+q+q^2+\cdots+q^{n-1},$$

because $\frac{1-q^n}{1-q}=1+q+q^2+\cdots+q^{n-1}$ almost everywhere.

A *q-factorial* of n, $\{n\}_q!$, is defined as a product of q-analogs of i, where $0<i\leq n$. Hence,

$$\begin{aligned}\{n\}_q! &= \{n\}_q\cdot\{n-1\}_q\cdots\{1\}_q\\ &= \tfrac{1-q^n}{1-q}\cdot\tfrac{1-q^{n-1}}{1-q}\cdots\tfrac{1-q}{1-q}\\ &\equiv (1+q+q^2+\cdots+q^{n-1})(1+q+q^2+\cdots+q^{n-2})\cdots(1+q)(1).\end{aligned}$$

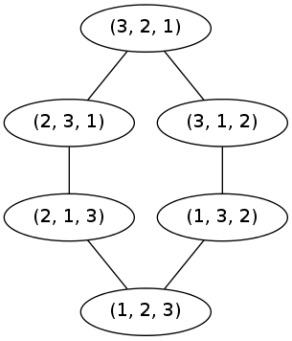
Similar as the binomial coefficient, the *q-binomial coefficient* is given by

$$\binom{n}{m}_q=\frac{\{n\}_q!}{\{m\}_q!\{n-m\}_q!}$$

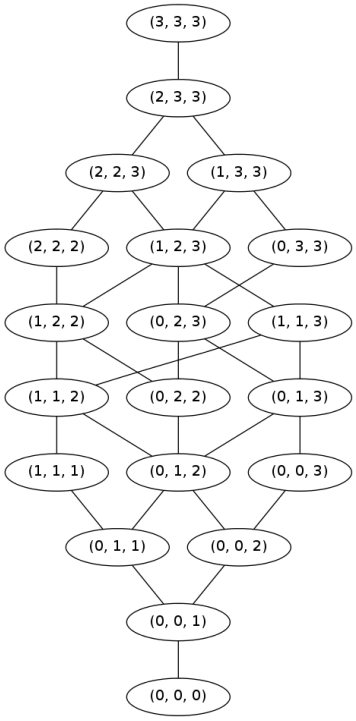
The *Boolean algebra* B_n is the poset of all subsets of a set with n elements, ordered by an inclusion. The size of B_n is 2^n and its Whitney numbers correspond to a row in the Pascal's triangle. The rank of an element is its cardinality.

The *division* poset D_n is a poset of all divisors of n, ordered by divisibility. Let $n=p_1^{a_1}p_2^{a_2}\ldots p_k^{a_k}$ be the prime factorization of n. Each element of the division poset can be written as $p_1^{b_1}p_2^{b_2}\ldots p_k^{b_k}$, where $0\leq b_i\leq a_i$ for each i. The rank of $p_1^{b_1}p_2^{b_2}\ldots p_k^{b_k}$ is $\sum_{i=1}^k b_i$.

The *Symmetric group* S_n is the group of all permutations of $(1,2,\ldots,n)$. The sum of all Whitney numbers of any partial ordering on S_n has to be $n!$ since there are $n!$ possible arrangements of n elements in a sequence. Suppose $a=(a_1,a_2,\ldots,a_n)$ and $b=(b_1,b_2,\ldots,b_n)$ are elements of S_n. A pair (i,j) is called an *inversion* of a if $i<j$



(a) Inversion poset I_3

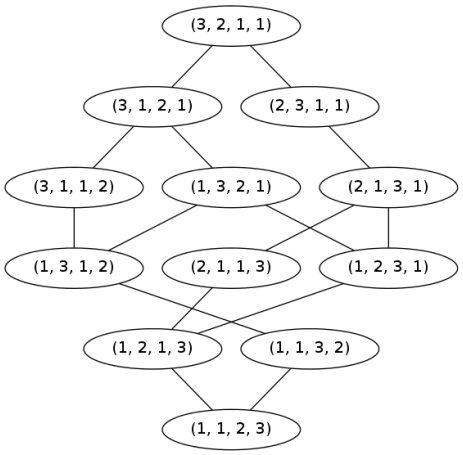


(b) Young lattice $L(3, 3)$

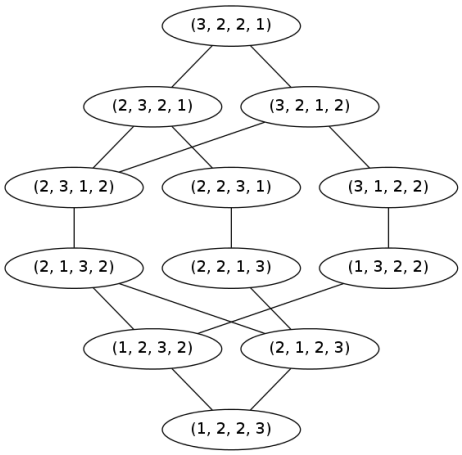
Figure 2.3: Diagrams of an inversion poset and a Young's lattice.

and $a_i > a_j$. The *inversion sequence* of a is a sequence $(i_1, i_2, \ldots, i_n)$ where each i_k represents how many elements in a on the left of a_k are greater than a_k [7]. For example, consider a sequence $(7, 5, 8, 3, 2, 6, 1, 4)$. A pair $(2, 7)$ is an inversion since $5 > 1$ and the inversion sequence is $(0, 1, 0, 3, 4, 2, 6, 4)$. The *inversion number* of a permutation is the sum of its inversion sequence, i.e., $\text{inv}(a) = \sum_{j=1}^{n} i_j$ [11]. The set S_n with a relation given by $a \lessdot b$, if b is created from a by interchanging a_i with a_{i+1} when $a_i < a_{i+1}$, is called the *inversion poset*, I_n. This binary relation is called the (weak) *Bruhat* order. The generating function is given by

$$GF(I_n) = \{n\}_q! = \prod_{i=0}^{n-1} (1 + q + q^2 + \cdots + q^i)$$



(a) Inversion poset $I_{\{1,1,2,3\}}$



(b) Inversion poset $I_{\{1,2,2,3\}}$

Figure 2.4: Diagrams of inversion posets for multisets.

and the rank of an element is its inversion number.

Similar to inversion posets we define an inversion poset for multisets ordered by inversions. Inversion posets for $\{1,1,2,3\}$ and $\{1,2,2,3\}$, $I_{\{1,1,2,3\}}$ and $I_{\{1,2,2,3\}}$, respectively, are shown in Figure 2.4. The generating function for I_N, where $N = \{\underbrace{n_1, n_1, \ldots, n_1}_{l_1}, \underbrace{n_2, n_2, \ldots, n_2}_{l_2}, \ldots, \underbrace{n_k, n_k, \ldots, n_k}_{l_k}\}$ is given by a q–multinomial coefficient,

$$GF(I_N) = \left(\begin{array}{c} \sum_{i=1}^{k} l_i \\ l_1, l_2, \ldots, l_k \end{array} \right)_q = \frac{\left\{ \sum_{i=1}^{k} l_i \right\}_q !}{\{l_1\}_q! \cdot \{l_2\}_q! \cdots \{l_k\}_q!}.$$

We will prove this result in Chapter 3. For example, for $I_{\{1,1,2,3\}}$ we have

$$\begin{aligned} GF(I_{\{1,1,2,3\}}) = & \frac{\{2+1+1\}_q!}{\{2\}_q!\{1\}_q!\{1\}_q!} = \frac{(1+q+q^2+q^3)(1+q+q^2)(1+q)(1)}{(1+q)(1)(1)} \\ = & (1+q+q^2+q^3)(1+q+q^2) \\ = & 1+2q+3q^2+3q^3+2q^4+q^5. \end{aligned}$$

Note that the generating function for $I_{\{1,2,2,3\}}$ is the same as the generating function for $I_{\{1,1,2,3\}}$ even though these posets are not isomorphic.

The *Young*'s lattice, $L(m,n)$, is a poset of n-tuples $(a_1, a_2, \ldots, a_n)$, where $0 \leq a_1 \leq a_2 \leq \cdots \leq a_n \leq m$ with order relation $(a_1, a_2, \ldots, a_n) \leq (b_1, b_2, \ldots, b_n)$ if $a_i \leq b_i$ for all i. The rank function is given by a sum of all elements in an n-tuple, hence $\rho((a_1, a_2, \ldots, a_n)) = a_1 + a_2 + \cdots + a_n$. We can find in [11, p. 72] that the generating function for a Young's lattice is

$$GF(L(m,n)) = \binom{m+n}{m}_q = \frac{(1-q^{n+1})(1-q^{n+2})\cdots(1-q^{n+m})}{(1-q)(1-q^2)\cdots(1-q^m)}.$$

Theorem 2.2 ([2])**.** *If $\mathcal{P}$ and $\mathcal{Q}$ are posets with a symmetric saturated chain decomposition, then $\mathcal{P} \times \mathcal{Q}$ has a symmetric saturated chain decomposition.*

Proof. Let $P_0, P_1, \ldots, P_m$ and $Q_0, Q_1, \ldots, Q_n$ be SSCD's for $\mathcal{P}$ and $\mathcal{Q}$, respectively. Consider a pair of chains (P_i, Q_j), say

$$\mathcal{P}_i = p_0 \lessdot p_1 \lessdot \cdots \lessdot p_k \text{ and } \mathcal{Q}_j = q_0 \lessdot q_1 \lessdot \cdots \lessdot q_h.$$

We can create a new chain E_l, where

$$E_l = \Big((p_0, q_l), (p_1, q_l), \ldots, (p_{k-l}, q_l), (p_{k-l}, q_{l+1}), \ldots, (p_{k-l}, q_h)\Big),$$

for $0 \leq l \leq \min\{k, h\}$. Each E_l is evidently a saturated chain in $\mathcal{P} \times \mathcal{Q}$, since the change is only in the first or the second term and both P_i and Q_j are saturated. $\mathcal{P}_i$ and $\mathcal{Q}_j$ are

symmetric, so $\rho(\mathcal{P}) = \rho(p_0) + \rho(p_k)$ and $\rho(\mathcal{Q}) = \rho(q_0) + \rho(q_h)$. Additionally, since

$$\begin{aligned}
\rho((p_0, q_l)) + \rho((p_{k-l}, q_h)) &= \rho(p_0) + \rho(q_l) + \rho(p_{k-l}) + \rho(q_h) \\
&= \rho(p_0) + \rho(q_k) - (k-l) + \rho(p_0) + (k-l) + \rho(q_h) \\
&= \rho(p_0) + \rho(q_k) + \rho(p_0) + \rho(q_h) \\
&= \rho(\mathcal{P}) + \rho(\mathcal{Q}) \\
&= \rho(\mathcal{P} \times \mathcal{Q}),
\end{aligned}$$

E_l is also symmetric. If $k \leq h$, the number of elements in all E_l's for a pair (P_i, Q_j) is

$$\begin{aligned}
\sum_{l=0}^{k} |E_l| &= \sum_{l=0}^{k} (k+h+1-2l) = (k+1)(k+h+1) - 2\sum_{l=0}^{k} l \\
&= k^2 + kh + 2k + h + 1 - 2k\tfrac{k+1}{2} \\
&= k^2 + kh + 2k + h + 1 - k(k+1) = kh + k + h + 1 \\
&= (k+1)(h+1) = |P_i| \cdot |Q_j|.
\end{aligned}$$

If $h < k$, the same result can be obtained by the same procedure. Since we can do this procedure for each pair of chains from the decomposition, the total number of elements in all E_l's is

$$\sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{l=0}^{\min\{k,h\}} |E_l| = \sum_{i=0}^{m} \sum_{j=0}^{n} |P_i| \cdot |Q_j| = \sum_{i=0}^{m} |P_i| \cdot \sum_{j=0}^{n} |Q_j| = |\mathcal{P}| \cdot |\mathcal{Q}|.$$

Hence we will obtain new symmetric chain decomposition of $\mathcal{P} \times \mathcal{Q}$ by this procedure. □

For example, consider a direct product of I_3 and a chain of three elements, $C = (c_1, c_2, c_3)$. A diagram of this direct product is shown in Figure 2.5. To follow the proof of Theorem 2.2 we pick a symmetric saturated chain in each poset. These chains are highlighted by thick solid line. These chains will yield, by Theorem 2.2, three symmetric saturated chains in the direct product, since the shortest chain contains three elements. If we follow the same notation, we have chains E_0, E_1 and E_2. The chain E_0 is the

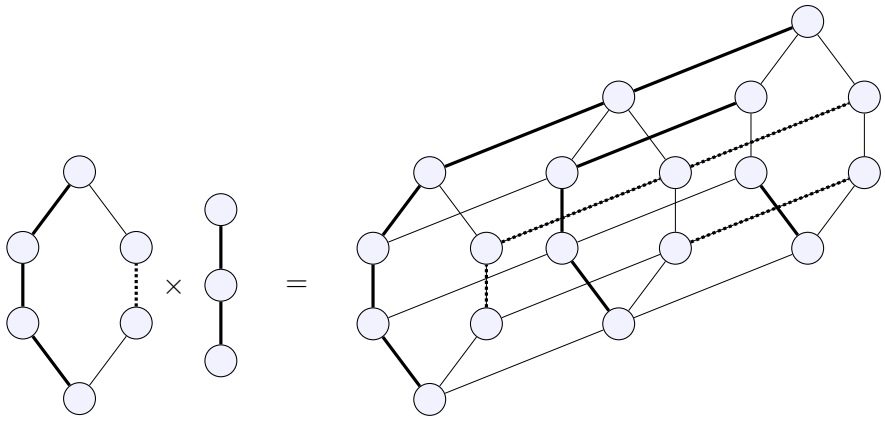


Figure 2.5: Example of theorem 2.2.

longest chain and E_2 is the shortest. Additionally, these chains cover all elements in the Cartesian product of the original highlighted chain in I_3 and C. We can describe this construction easier if a diagram of the direct product is available. Given two saturated symmetric chains $\mathcal{P}$ and $\mathcal{Q}$, each from a different poset, a SSCD of the direct product $\mathcal{P} \times \mathcal{Q}$ can be found by following an algorithm:

1. Let $S = \mathcal{P} \times \mathcal{Q}$.
2. Pick an element p with the smallest rank in S.
3. Construct a new symmetric saturated chain E_i which starts at p, follows an upward path given by $\mathcal{P}$ as far as it can and then continues to the right following a path given by $\mathcal{Q}$.
4. If we have not used all elements in $\mathcal{P} \times \mathcal{Q}$ we have to shrink S and find a next chain. In order to do that we have to remove E_i from S and go back to 2.

We will investigate a few types of posets which are created as a direct product of two or more ranked posets. To analyze their properties we prove the following theorems.

Theorem 2.3 ([4])**.** *The generating function for the direct product of two ranked posets*

$\mathcal{P}$ and $\mathcal{Q}$ is given by

$$GF(\mathcal{P} \times \mathcal{Q}) = GF(\mathcal{P}) \cdot GF(\mathcal{Q}).$$

Proof. Since the generating function of a poset $\mathcal{P}$ is given by $GF(\mathcal{P}) = \sum_{p \in \mathcal{P}} q^{\rho(p)}$, we can use this formula on $\mathcal{P} \times \mathcal{Q}$

$$\begin{aligned} GF(\mathcal{P} \times \mathcal{Q}) &= \sum_{(a,b) \in \mathcal{P} \times \mathcal{Q}} q^{\rho((a,b))} = \sum_{(a,b) \in \mathcal{P} \times \mathcal{Q}} q^{\rho(a)+\rho(b)} = \sum_{a \in \mathcal{P}} \sum_{q \in \mathcal{Q}} q^{\rho(a)+\rho(b)} \\ &= \sum_{a \in \mathcal{P}} q^{\rho(a)} \sum_{b \in \mathcal{Q}} q^{\rho(b)} = GF(\mathcal{P}) \cdot GF(\mathcal{Q}). \end{aligned}$$

□

Theorem 2.4. *A poset with a symmetric saturated chain decomposition has the Sperner property.*

Proof. If a poset $\mathcal{P}$ has a SSCD then the largest Whitney number is $W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P})$. We need to prove that every antichain has size less than or equal to $W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P})$.

Let $A = \{a_1, a_2, \ldots, a_l\}$ be an antichain and $C = \{C_1, C_2, \ldots, C_k\}$ be a SSCD of $\mathcal{P}$. Since C contains all elements of $\mathcal{P}$, suppose, without loss of generality that $a_1 \in C_1$. All elements in C_1 are comparable with a_1. Hence, without loss of generality, say that $a_2 \in C_2$. Since a_3 is not comparable with neither a_1 nor a_2 it cannot be in C_1 or C_2, say a_3 is in C_3. We can conclude, by reordering the C's as necessary, that $a_i \in C_i$ for each $i \leq l$, therefore $l \leq k$.

It is clear from the definition of SSCD that the biggest Whitney number, the size of the middle layer(s), is equal to the number of chains in a SSCD. Therefore,

$$l \leq k = W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}) \quad \Longrightarrow \quad l \leq W_{\lfloor \frac{\rho(\mathcal{P})}{2} \rfloor}(\mathcal{P}),$$

which completes the proof. □

Chapter 3

Additional properties of posets

3.1 Division and Boolean posets

Theorem 3.1. *A division poset D_n is isomorphic to the direct product of chains $C = (1 \lessdot p_1 \lessdot p_1^2 \lessdot \cdots \lessdot p_1^{a_1}) \times (1 \lessdot p_2 \lessdot p_2^2 \lessdot \cdots \lessdot p_2^{a_2}) \times \cdots \times (1 \lessdot p_k \lessdot p_k^2 \lessdot \cdots \lessdot p_k^{a_k})$, where $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ is the prime factorization of n.*

Proof. Let I be a function from D_n to C given by

$$I(p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}) = (p_1^{b_1}, p_2^{b_2}, \dots, p_k^{b_k}).$$

It is clear that the function I is one-to-one. Let $p = p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}$ and $q = p_1^{c_1} p_2^{c_2} \dots p_k^{c_k}$, where $b_i \leq c_i$ for each i. Apparently, q is divisible by p, $q/p = p_1^{c_1 - b_1} \cdot p_2^{c_2 - b_2} \dots p_k^{c_k - b_k}$ where each $c_i - b_i \geq 0$ and therefore $p \leq_{D_n} q$.

Since $I(p) = (p_1^{b_1}, p_2^{b_2}, \dots, p_k^{b_k})$ we can construct a chain $\Big((p_1^{b_1}, p_2^{b_2}, \dots, p_k^{b_k}) \leq (p_1^{c_1}, p_2^{b_2}, \dots, p_k^{b_k}) \leq (p_1^{c_1}, p_2^{c_2}, \dots, p_k^{b_k}) \leq \cdots \leq (p_1^{c_1}, p_2^{c_2}, \dots, p_k^{c_k}) = I(q)\Big)$ in C, therefore $I(p) \leq_C I(q)$ from the transitive property of the relation.

To finish the proof we need to show that $I(p) \leq_C I(q)$ implies $p \leq_{D_n} q$. To show this, again suppose that $I(p) = (p_1^{b_1}, p_2^{b_2}, \dots, p_k^{b_k})$ and $I(q) = (p_1^{c_1}, p_2^{c_2}, \dots, p_k^{c_k})$, where $b_i \leq c_i$ for each i. It is clear, since I is one-to-one, that p divides q. □

We can find the generating function for a division poset by combining Theorems 3.1 and 2.3.

Corollary 3.2. *The generating function for a division poset D_n, where $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ is the prime factorization of n, is given by*

$$GF(D_n) = \prod_{i=1}^{k} (1 + q + q^2 + \cdots + q^{a_i}).$$

For example the generating function for D_{36} is

$$GF(D_{36}) = GF(D_{2^2 \cdot 3^2}) = (1 + q + q^2) \cdot (1 + q + q^2) = 1 + 2q + 3q^2 + 2q^3 + q^4.$$

We can compare this rank function with a diagram of the division poset D_{36} in Figure 1.1(b), there is one element with rank zero, two with rank one and three with rank two.

Now, we may focus on Boolean posets and show that a Boolean poset has the same properties as a division poset. In fact, a Boolean poset is a special case of a division poset D_n.

Theorem 3.3. *A Boolean poset B_k is isomorphic to D_n, where $n = p_1 p_2 \cdots p_k$ and each p_i is a distinct prime number.*

Proof. Since elements of B_k are subsets of $\{1, 2, \dots, k\}$, we can introduce an one-to-one function I from B_k into D_n. Say

$$I(\{a_1, a_2, \dots, a_i\}) = p_{a_1} p_{a_2} \cdots p_{a_i},$$

where $I(\emptyset) = 1$. Consider two subsets of B_k, A and B, such that $A \subseteq B$. Suppose that $A = \{a_1, a_2, \dots, a_i\}$ and $B = \{a_1, a_2, \dots, a_i, b_1, b_2, \dots, b_j\}$ since A is a subset of B. We have to show that $I(A) \leq_{D_n} I(B)$, i.e., that the function I respects the order. Observe that

$$I(A) = p_{a_1} p_{a_2} \cdots p_{a_i} = \frac{p_{a_1} p_{a_2} \cdots p_{a_i} p_{b_1} p_{b_2} \cdots p_{b_j}}{p_{b_1} p_{b_2} \cdots p_{b_j}} = \frac{I(B)}{p_{b_1} p_{b_2} \cdots p_{b_j}}.$$

Thus

$$\frac{I(B)}{I(A)} = p_{b_1} p_{b_2} \cdots p_{b_j},$$

which indicates that $I(A)$ divides $I(B)$, therefore $I(A) \leq_{D_n} I(B)$. □

Now we know that a Boolean poset is a ranked poset with a SSCD. Using Corollary 3.2, the generating function for B_n is given by

$$GF(B_n) = \prod_{i=1}^{n} (1+q) = \sum_{i=0}^{n} \binom{n}{i} q^i,$$

which explains why a sequence of Whitney numbers corresponds to a row in the Pascal's triangle.

Theorem 3.4. *Given a division poset D_n, where $n = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$ is the prime factorization of n, we can find an isomorphism I between D_n with a standard order and D_n with a reversed order.*

Proof. We will prove that this isomorphism is given by

$$I(p_1^{b_1} p_2^{b_2} \ldots p_k^{b_k}) = \frac{n}{p_1^{b_1} p_2^{b_2} \ldots p_k^{b_k}}.$$

The function I is one-to-one, therefore we have to check only that if $b \leq_{D_n} c$ then $I(b) \geq_{D_n} I(c)$. Since c is divisible by b we have

$$\frac{c}{b} = \frac{\frac{1}{b}}{\frac{1}{c}} = \frac{\frac{1}{b}}{\frac{1}{c}} \cdot \frac{n}{n} = \frac{\frac{n}{b}}{\frac{n}{c}} = \frac{I(b)}{I(c)},$$

as required. □

Given a poset P, the poset with the same underlying set but reversed order is called the *dual* of P, P^* [4]. If a poset and its dual are isomorphic, then this poset is called *self–dual* [10].

Theorems 2.2 and 3.1 imply that each division poset has a SSCD, therefore each division poset has the Sperner property by Theorem 2.4. In particular, the Boolean poset B_n satisfies the Sperner property, proving Theorem 2.1.

3.2 Inversion posets and Young's lattices

Let $S(m+n,m)$ be a poset of all bit strings of length $m+n$ with m ones. We define a partial order which is the same as for an inversion poset I_n. That is, $p \leq q$ if q is created by some number of adjacent transpositions of elements in p, where the second element is larger than the first. The smallest and the greatest elements are $\underbrace{0000\cdots000}_{n}\underbrace{11\cdots111}_{m}$ and $\underbrace{11\cdots111}_{m}\underbrace{0000\cdots000}_{n}$, respectively.

Theorem 3.5. *$S(m+n,m)$ and $L(m,n)$ are isomorphic.*

Proof. Let I be a function from $L(m,n)$ to $S(m+n,m)$ given by

$$I\Big((a_1,a_2,\ldots,a_n)\Big) = \underbrace{1\cdots1}_{a_1}0\underbrace{1\cdots1}_{a_2-a_1}0\underbrace{1\cdots1}_{a_3-a_2}0\cdots\ \underbrace{1\cdots1}_{a_n-a_{n-1}}0\underbrace{1\cdots1}_{m-a_n}$$

To show that I is one-to-one suppose that $a=(a_1,a_2,\ldots,a_n), b=(b_1,b_2,\ldots,b_n)$ and $I(a)=I(b)$. Then

$$\begin{aligned} I(a) &= I(b) \\ \underbrace{1\cdots1}_{a_1}0\underbrace{1\cdots1}_{a_2-a_1}0\underbrace{1\cdots1}_{a_3-a_2}0\cdots\ \underbrace{1\cdots1}_{a_n-a_{n-1}}0\underbrace{1\cdots1}_{m-a_n} &= \underbrace{1\cdots1}_{b_1}0\underbrace{1\cdots1}_{b_2-b_1}0\underbrace{1\cdots1}_{b_3-b_2}0\cdots\ \underbrace{1\cdots1}_{b_n-b_{n-1}}0\underbrace{1\cdots1}_{m-a_n}. \end{aligned}$$

If the two strings are the same, then the number of ones separated by a zero has to be

the same. Hence, we get this system of equations

$$\begin{aligned} a_1 &= b_1 \\ a_2 - a_1 &= b_2 - b_1 \\ a_3 - a_2 &= b_3 - b_2 \\ &\vdots \\ a_n - a_{n-1} &= b_n - b_{n-1} \\ m - a_n &= m - b_n. \end{aligned}$$

We can use the solution of the first equation, $a_1 = b_1$, to solve the second equation and get $a_2 = b_2$. It is easy to see that the solution of this system is $a_i = b_i$ for $1 \leq i \leq n$ and therefore $a = b$.

To show that the function I preserves order, let $a \leq_L b$. $I(a)$ is less than or equal to $I(b)$ if the number of ones on the left from each zero in $I(a)$ is less than or equal to the number of ones on the left from each zero in $I(b)$. For $I(a)$, the number of ones before the first zero is a_1, before the second zero it is $a_1 + a_2 - a_1 = a_2$, there are a_3 ones before the third zero, etc., up to m. Similarly, the sequence for b is $b_1, b_2, \ldots, m$. Each a_i is less than or equal to b_i since $a \leq_L b$ and therefore $I(a) \leq I(b)$.

The opposite, $I(a) \leq I(b)$ implies $a \leq_L b$ comes from the same effort of comparing number of ones which precede a zero. □

The function I from the proof above is one-to-one and therefore it has an inverse. The function I^{-1} takes a bit string s and returns a vector containing only those elements from the inversion sequence of s which correspond to zeros in s. An example of an isomorphism between $L(2,3)$ and $S(5,2)$ can be seen on Figure 3.1.

Kyle Krueger in his Master's project[8] proved that $\binom{m+n}{m}_q$ is the generating function of $S(m+n, m)$. This result together with the previous theorem shows that the generation function for $L(m,n)$ is

$$GF(L(m,n)) = \binom{m+n}{m}_q.$$

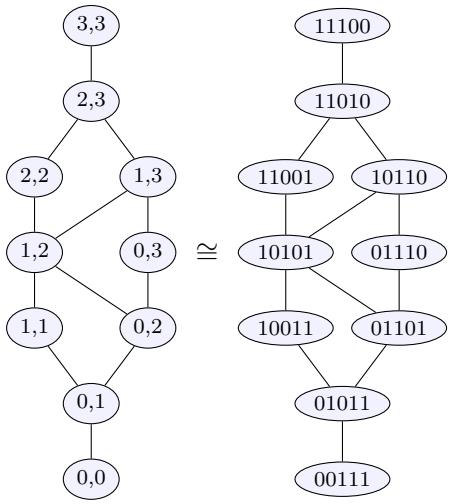


Figure 3.1: An isomorphism between $L(3,2)$ and $S(5,3)$.

Theorem 3.6. *The generating function for an inversion poset for a multiset I_N, where $N = \{\underbrace{n_1, n_1, \ldots, n_1}_{l_1}, \underbrace{n_2, n_2, \ldots, n_2}_{l_2}, \ldots, \underbrace{n_k, n_k, \ldots, n_k}_{l_k}\}$ is*

$$\begin{pmatrix} \sum_{i=1}^{k} l_i \\ l_1, l_2, \ldots, l_k \end{pmatrix}_q.$$

Proof. We prove this theorem using strong induction on k, the number of distinct elements in N. Let $k = 2$ be our base case. Krueger in [8] provides the formula for the generating function in this base case as a q-binomial coefficient,

$$\binom{l_1 + l_2}{l_1}_q = \frac{\{l_1 + l_2\}_q!}{\{l_1\}_q! \cdot \{l_2\}_q!}.$$

This is the same formula that we get from our hypothesis, since

$$\begin{pmatrix} \sum_{i=1}^{2} l_i \\ l_1, l_2 \end{pmatrix}_q = \frac{\{l_1 + l_2\}_q!}{\{l_1\}_q! \cdot \{l_2\}_q!}.$$

Assume that the formula holds for each k between 1 and m. The generating function for I_N, where $N = \{\underbrace{n_1, n_1, \ldots, n_1}_{l_1}, \underbrace{n_2, n_2, \ldots, n_2}_{l_2}, \ldots, \underbrace{n_m, n_m, \ldots, n_m}_{l_m}\}$, can be found in a few steps. First, let us recall a definition of the generating function for the inversion poset

$$GF(I_N) = \sum_{w \in I_N} q^{\mathrm{inv}(w)},$$

where the inversion number of w is a sum of its inversion sequence. Additionally, we introduce a bijection $\varphi : I_N \to W' \times W''$, where $\varphi(w) = (w', w'')$. The permutation w' is a copy of w without any n_m's and w'' is a copy of w where each element smaller than n_m is replaced by zero. For example if $N = \{1, 1, 2, 2, 2, 3, 3, 4, 4\}$ and $w = (3, 1, 2, 4, 1, 2, 4, 3, 2)$ then $w' = (3, 1, 2, 1, 2, 3, 2)$, $w'' = (0, 0, 0, 4, 0, 0, 4, 0, 0)$, $\mathrm{inv}(w) = 13$, $\mathrm{inv}(w') = 7$ and $\mathrm{inv}(w'') = 6$.

This bijection splits the inversion number of w into two parts. The inversion number of w'' expresses the contribution of the n_m's to the inversion number of w and the inversion number of w' indicates the contribution of all other elements. We can represent this relation by the following equation,

$$\mathrm{inv}(w) = \mathrm{inv}(w') + \mathrm{inv}(w'').$$

It is apparent that φ is an injective function. To show that it is also surjective, given a pair w', w'' we can find w by replacing all zeros in w'' with elements from w' in a given order.

This bijection allows to rewrite the generating function for I_N,

$$\begin{aligned}\sum_{w \in I_N} q^{\mathrm{inv}(w)} &= \sum_{\substack{w \in I_N \\ (w', w'') = \varphi(w)}} q^{\mathrm{inv}(w') + \mathrm{inv}(w'')} = \sum_{(w', w'') \in W' \times W''} q^{\mathrm{inv}(w')} \cdot q^{\mathrm{inv}(w'')} \\ &= \sum_{w' \in W'} q^{\mathrm{inv}(w')} \cdot \sum_{w'' \in W''} q^{\mathrm{inv}(w'')}.\end{aligned}$$

Since W' contains all permutations of $\{\underbrace{n_1,\ldots,n_1}_{l_1},\underbrace{n_2\ldots,n_2}_{l_2},\ldots,\underbrace{n_{m-1},\ldots,n_{m-1}}_{l_{m-1}}\}$, the generating function will be the same as for $I_{\{\underbrace{n_1,\ldots,n_1}_{l_1},\underbrace{n_2\ldots,n_2}_{l_2},\ldots,\underbrace{n_{m-1},\ldots,n_{m-1}}_{l_{m-1}}\}}$ and therefore by the induction assumption,

$$\sum_{w'\in W'} q^{\mathrm{inv}(w')} = \begin{pmatrix} \sum_{i=1}^{m-1} l_i \\ l_1, l_2, \ldots, l_{m-1} \end{pmatrix}_q = \frac{\left\{\sum_{i=1}^{m-1} l_i\right\}_q!}{\{l_1\}_q!\cdot\{l_2\}_q!\cdots\{l_{m-1}\}_q!}.$$

Similarly, W'' contains all permutations of $\{\underbrace{0,0,\ldots,0,0}_{l_1+l_2+\cdots+l_{m-1}},\underbrace{n_m,\ldots,n_m}_{l_m}\}$ and therefore the generating function will be the same as the generating function for $I_{\{\underbrace{0,0,\ldots,0,0}_{l_1+l_2+\cdots+l_{m-1}},\underbrace{n_m,\ldots,n_m}_{l_m}\}}$. This function is our base case, thus

$$\sum_{w''\in W''} q^{\mathrm{inv}(w'')} = \begin{pmatrix} \sum_{i=1}^{m-1} l_i + l_m \\ \sum_{i=1}^{m-1} l_i \end{pmatrix}_q = \frac{\left\{\sum_{i=1}^{m} l_i\right\}_q!}{\left\{\sum_{i=1}^{m-1} l_i\right\}_q!\cdot\{l_m\}_q!}.$$

Using the general definition of a generating function for an inversion poset we can describe the generating function for $I_{\{\underbrace{n_1,n_1,\ldots,n_1}_{l_1},\underbrace{n_2,n_2,\ldots,n_2}_{l_2},\ldots,\underbrace{n_m,n_m,\ldots,n_m}_{l_m}\}}$ as a product of the q-multinomial coefficient and the q-binomial coefficient,

$$\begin{aligned} \frac{\left\{\sum_{i=1}^{m-1} l_i\right\}_q!}{\{l_1\}_q!\cdot\{l_2\}_q!\cdots\{l_{m-1}\}_q!}\cdot\frac{\left\{\sum_{i=1}^{m} l_i\right\}_q!}{\left\{\sum_{i=1}^{m-1} l_i\right\}_q!\cdot\{l_m\}_q!} &= \frac{\left\{\sum_{i=1}^{m} l_i\right\}_q!}{\{l_1\}_q!\cdot\{l_2\}_q!\cdots\{l_m\}_q!} \\ &= \begin{pmatrix} \sum_{i=1}^{m} l_i \\ l_1, l_2, \ldots, l_m \end{pmatrix}_q, \end{aligned}$$

as required. □

Theorem 3.7. *An inversion poset for a multiset I_N is self–dual.*

Proof. Let $F : I_N \to I_N^*$ be given by

$$F\Big((p_1, p_2, \ldots, p_m)\Big) = (p_m, p_{m-1}, \ldots, p_2, p_1).$$

Let a and b be two permutations of I_N such that $F(a) = F(b)$. Elements a and b must be the same, since their reverses are the same and therefore F is one-to-one.

Let $a \leq b$, thus b can be constructed from a by some number of swaps of consecutive elements where the element on the right is greater than the element on the left. Let $C = (a, c_1, c_2, \ldots, c_m, b)$ be a saturated chain between a and b. Since $c_m \lessdot b$, there has to be a pair of consecutive elements in b where the element on the right is less than the element on the left. Thus, we know that $F(b) \lessdot F(c_m)$ since F reverses the order of b and c_m. Similarly, we can show that $F(c_m) \lessdot F(c_{m-1}) \lessdot \cdots \lessdot F(c_1) \lessdot F(a)$ and therefore $F(b) \leq F(a)$. □

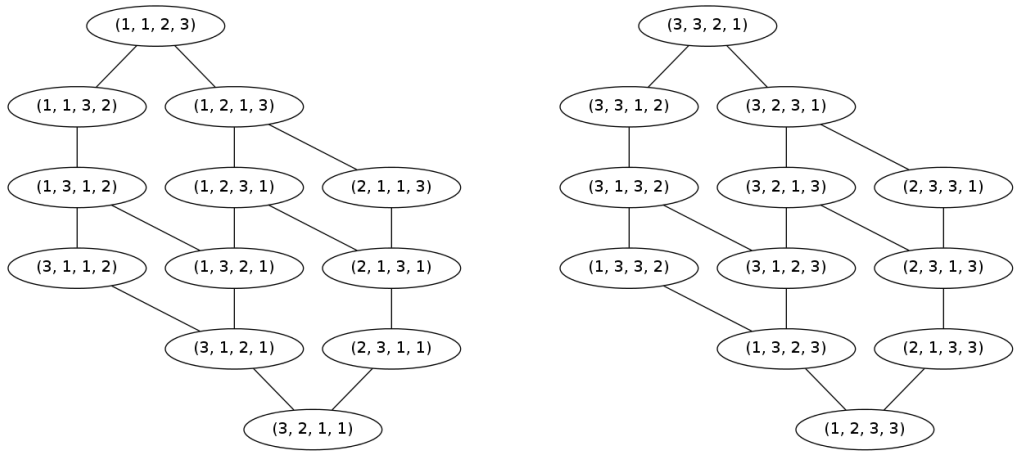
The next theorem introduces another isomorphism between inversion posets which is convenient when a SSCD is being found.

Theorem 3.8. *The inversion poset $I_{\{\underbrace{n_1,\ldots,n_1}_{l_1},\underbrace{n_2,\ldots,n_2}_{l_2},\ldots,\underbrace{n_k,\ldots,n_k}_{l_k}\}}$ is isomorphic to the dual of the inversion poset $I_{\{\underbrace{n_1,\ldots,n_1}_{l_k},\underbrace{n_2,\ldots,n_2}_{l_{k-1}},\ldots,\underbrace{n_k,\ldots,n_k}_{l_1}\}}$.*

Proof. For the sake of simplicity let $\overline{n_i} = n_{k-i+1}$ and $m = \sum_{i=1}^{k} l_i$. Let $F : I_{N_1} \to I_{N_2}^*$ be a function given by

$$F\Big((p_1, p_2, \ldots, p_m)\Big) = (\overline{p_1}, \overline{p_2}, \ldots, \overline{p_m}),$$

where $N_1 = \{\underbrace{n_1,\ldots,n_1}_{l_1}, \underbrace{n_2,\ldots,n_2}_{l_2}, \ldots, \underbrace{n_k,\ldots,n_k}_{l_k}\}$ and $N_2 = \{\underbrace{n_k,\ldots,n_k}_{l_1}, \underbrace{n_{k-1},\ldots,n_{k-1}}_{l_2}, \ldots, \underbrace{n_1,\ldots,n_1}_{l_k}\}$. Consider two permutations a and b of I_{N_1} such that $F(a) = F(b)$. It is clear from the definition of F that since $F(a) = F(b)$ then $a = b$ and therefore F is one-to-one.



(a) The dual of the inversion poset $I_{\{1,1,2,3\}}$.

(b) The inversion poset $I_{\{1,2,3,3\}}$.

Figure 3.2: An illustration of an isomorphism between $I_{\{1,1,2,3\}}$ and $I_{\{1,2,3,3\}}$.

To prove the isomorphism we have to show that $a \leq b$ if and only if $F(a) \geq F(b)$. Since we showed that F is one-to-one it is enough to show that $a < b$ if and only if $F(a) > F(b)$. To show this, we will prove that $a \lessdot b$ if and only if $F(a) \gtrdot F(b)$ because these statements are equivalent since there is always a saturated chain between a and b if $a < b$.

Let $a = (a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_m)$, where $a_i < a_{i+1}$ and suppose that $a \lessdot b$, with $b = (a_1, a_2, \ldots, a_{i+1}, a_i, \ldots, a_m)$.

$$F(b) = (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_{i+1}}, \overline{a_i}, \ldots, \overline{a_m})$$

Let's focus on the pair $(\overline{a_{i+1}}, \overline{a_i})$. Since $a_i < a_{i+1}$ we can say that $n_{j_1} = a_i < a_{i+1} = n_{j_2}$, where $j_1 < j_2$. Then $\overline{a_i} = \overline{n_{j_1}} = n_{k-j_1+1}$ and $\overline{a_{i+1}} = \overline{n_{j_2}} = n_{k-j_2+1}$. It follows from the inequality $j_1 < j_2$ that $k - j_2 + 1 < k - j_1 + 1$, $n_{k-j_2+1} < n_{k-j_1+1}$ and hence $\overline{a_{i+1}} < \overline{a_i}$. Therefore, we can conclude that

$$F(b) = (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_{i+1}}, \overline{a_i}, \ldots, \overline{a_m}) \lessdot (\overline{a_1}, \overline{a_2}, \ldots, \overline{a_i}, \overline{a_{i+1}}, \ldots, \overline{a_m}) = F(a),$$

as required.

If we follow previous steps in the opposite direction we get that $F(a) \succcurlyeq F(b)$ implies $a \preccurlyeq b$, which completes the proof. □

The previous theorem implies that $I_{\{1,1,2,3\}}$ is isomorphic to the dual of $I_{\{1,2,3,3\}}$. Additionally, the dual of the inversion poset $I_{\{1,2,3,3\}}$ is isomorphic to $I_{\{1,2,3,3\}}$, because the inversion poset is self–dual. Therefore, Theorems 3.7 and 3.8 yield the following corollary.

Corollary 3.9. *The inversion poset* $I_{\{\underbrace{n_1,\ldots,n_1}_{l_1},\underbrace{n_2,\ldots,n_2}_{l_2},\ldots,\underbrace{n_k,\ldots,n_k}_{l_k}\}}$ *is isomorphic to the inversion poset* $I_{\{\underbrace{n_1,\ldots,n_1}_{l_k},\underbrace{n_2,\ldots,n_2}_{l_{k-1}},\ldots,\underbrace{n_k,\ldots,n_k}_{l_1}\}}$.

Chapter 4

The algorithm

We described, with a proof, an algorithm for finding a SSCD for a division poset in Chapter 3. We introduced Inverse posets and Young's lattices but we didn't indicate any decomposition for these posets. The reason for this is that there is no general proof that these posets have a SSCD. Stanley gave a combinatorial proof that a Young's lattice, $L(m, n)$, is rank unimodal and has the Sperner property in [9]. Although, it was proven by Sylvester in 1878 that $L(m, n)$ is rank unimodal, Stanley gave the first combinatorial proof. He also conjectured that $L(m, n)$ has a SSCD for every positive m and n. This conjecture has not yet been proven for a general case, only cases where $\min(m, n) \leq 4$ were proven [3].

We will introduce an enhanced algorithm for finding a SSCD for a general ranked poset $\mathcal{P}$ based on the previous work of Katsumata in [7]. The algorithm starts with $W_{\lfloor \rho(\mathcal{P})/2 \rfloor}$ chains located in the middle of $\mathcal{P}$ since the center of every symmetric saturated chain is located in the middle layer(s). It will create chains of length 0 or 1 based on the number of layers and then it will augment some of these chains in a way that they will remain symmetric and saturated.

Our algorithm differs significantly from the algorithm provided by Katsumata, since it uses a flow algorithm for the augmentation of chains and it can work with any poset

given by its adjacency list, not just with an inversion poset I_n. If the program finds a SSCD of the poset, it saves this decomposition into a text file and creates four pictures of Hasse diagrams with highlighted chains. Pictures for selected posets are displayed in appendix B.

4.1 Finding a flow in a graph

A directed graph can represent many structures from everyday life, like public transportation, supply chains, road maps, computer networks or simplified water supply system. Each of these structures can be represented as a system of pipes, connected with joints, with a flow of an incompressible fluid. We will focus on a system with only one point of entry and only one exit point for the fluid and we will try to investigate what the maximal flow through the system is.

This abstraction gives rise to three constraints on the flow based on physics. There is a limitation on how fast we can transport the fluid through a pipe based on the material of the pipe, its radius and a viscosity of the fluid and other factors. This limitation is called a capacity and each pipe can have a different capacity. The next constraint is called antisymmetry and it means that a flow in a pipe is uniform – the flow is always in only one direction. The last constraint is that fluid can leave the system only at one joint and it can enter only at one joint and these joints have to be different. The exact definitions follow.

Let a directed graph $G = (V, E)$ with a capacity function $c : V^2 \to \mathbb{R}_0^+$, where $c(e) = 0$ if $e \notin E$, is given. Furthermore, denote sizes of V and E by n and m, respectively. A *flow* between two distinct vertices, source s and sink t, is given by a function $f : V^2 \to \mathbb{R}_0^+$ such that $\forall e \in E$:

$$f(e) \leq c(e) \qquad \text{(capacity constraint)}, \tag{4.1}$$

$$f((u, v)) = -f((v, u)) \text{ where } (u, v) = e \quad \text{(antisymmetry constraint)}, \tag{4.2}$$

$$\sum_{(u,v)\in V\setminus\{s,t\}} f((u,v)) = 0 \qquad \text{(flow conservation constraint).} \tag{4.3}$$

The *value* $|f|$ is defined as the total flow into the sink, i.e. $|f| = \sum_{u\in V} f((u,t))$. A *maximum flow* is a flow of maximum value. We define a *residual capacity* $c_r(e)$ to be a difference between a capacity of an edge and a flow through an edge, $c_r(e) = c(e) - f(e)$. An edge is *saturated* if its residual capacity is zero.

Many algorithms are available for finding a maximum flow in a graph [5]. We will focus on two types of flow algorithms, on the Ford–Fulkerson algorithm and the Push–Relabel algorithm.

The Ford–Fulkerson algorithm

This algorithm was first published in 1956 and improved versions of this algorithm were published by Dinic in 1970 and by Edmonds and Karp in 1972 [5]. This algorithm works as follows:

1. $f((u,v)) \leftarrow 0$ for every pair of vertices in V
2. while there exist a path p between s and t such that $c_r(e) > 0$ for every edge in this path p:
   (a) denote the smallest residual capacity in p by r
   (b) for each edge (u,v) in p:
      i. $f((u,v)) \leftarrow f((u,v)) + r$
      ii. $f((v,u)) \leftarrow f((v,u)) - r$

The Ford–Fulkerson algorithm finds a path through unsaturated edges in each iteration and it adds this path to the flow. This path is frequently called an augmenting path and the performance of this algorithm depends on an algorithm used for finding this path. Finding a path in a graph is generally implemented as a depth–first or breadth–first search on vertices. Both of these methods have their advantages. The Ford–Fulkerson

algorithm has a running time of $O(n^2m)$ or $O(nm^2)$ depending on whether a breadth–first or a depth–first search is used, respectively [6].

Using the Ford–Fulkerson algorithm, we have to find a new path between s and t in every cycle. This disadvantage is addressed in the next maximum flow algorithm.

The Push–Relabel algorithm

This algorithm does not find only one path from the source to the sink as the Ford–Fulkerson algorithm, but it pushes through the graph as much flow as is possible. For this purpose it manages a preflow and a vertex labeling d. *Preflow* is a function from V^2 to $\mathbb{R}$ which satisfies equations (4.1) and (4.2) for every edge in the graph. Preflow does not have to conserve the flow, it allows an incoming flow into a vertex to be bigger then an outgoing flow from the vertex. The algorithm starts with a preflow and it improves it into a maximal flow. Any valid preflow can be used, but commonly a preflow f_p where $f_p((s,p)) = c((s,p))$ for every $(s,p) \in E$, where s is the source, and $f_p(e) = 0$ for other edges is used.

After an initialization the algorithm either pushes a flow through a vertex or it updates the labeling. This vertex labeling provides estimates for a distance from the source and a distance to the sink for each vertex. The first implementation of this algorithm, by Goldberg in 1985, had a running time of $O(n^3)$ [6]. One year later, Goldberg and Tarjan published an implementation with a running time of $O(nm \log(n^2/m))$ [5].

4.2 Algorithm description

Our algorithm, similar to an algorithm of Katsumata, is implemented in three stages. These stages are making a poset, finding a SSCD and arrangement of chains.

1. Firstly, we create the Hasse diagram of a given poset or we read the poset structure from files. We represent the poset as a directed graph, e.g. as a list of vertices and an adjacency list, V and E, respectively. This graph can be partitioned into

levels since we are representing the Hasse diagram and we store these levels in another list to improve the performance. Denote the rank of the poset by k and let $N = \{N_0, N_1, \ldots, N_k\}$ be the collection of level sets. Additionally, we will create an empty list C for storing a SSCD.

When the first stage is finished, the list N has $k+1$ elements in it: $N_0, N_1, \ldots, N_k$. Each of these lists contains vertices with the same rank.

2. If k is odd, there are two middle levels in the poset, we have to find a perfect matching between them. This matching yields the two middle elements from each chain and it can be found by making a graph H. This graph contains vertices from $N_{(k-1)/2}$, $N_{(k+1)/2}$ and two other vertices s and t. We add all edges between $N_{(k-1)/2}$ and $N_{(k+1)/2}$ from E to H, connect s with every vertex from $N_{(k-1)/2}$ and connect each vertex from $N_{(k+1)/2}$ to t. Finding a maximum flow in H between s and t, where the capacity of every edge is 1, yields the matching. If the flow value is equal to $|N_{(k-1)/2}|$, we erase edges with no flow in H, all incidence edges of s, t and the vertices s and t. Edges which are left are saved into C. If the value is smaller, the poset does not have a SSCD.

   If k is even, we fill C with vertices from $N_{k/2}$.

   For $i = \lfloor k/2 \rfloor, \lfloor k/2 \rfloor - 1, \lfloor k/2 \rfloor - 2, \ldots, 1$ perform following:

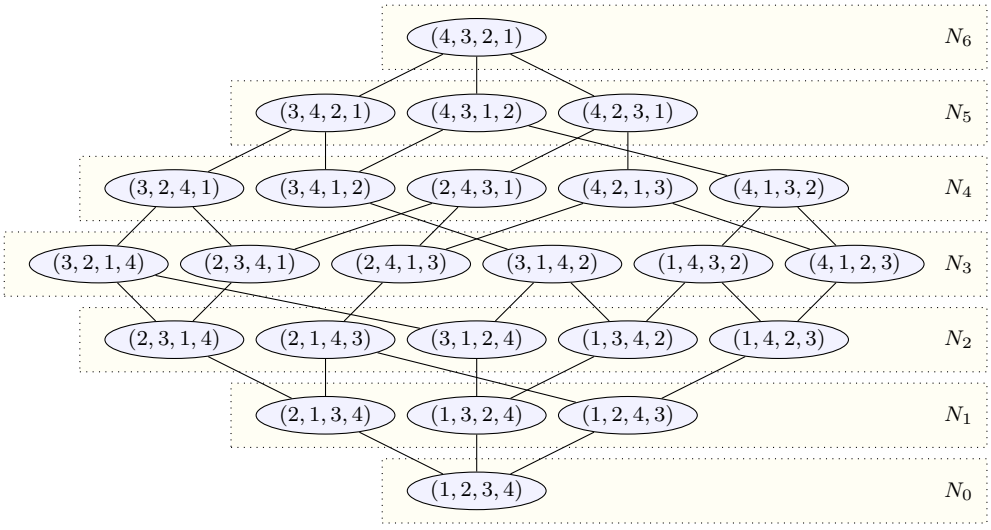
   (a) Make a graph H which contains vertices from N_{i-1}, N_i, N_{k-i}, $N_{k-(i-1)}$ and two extra vertices s and t. Connect s with each vertex in N_{i-1} and each vertex in $N_{k-(i-1)}$ with t. If there is an edge in E between vertices from N_{i-1} and N_i, connect the corresponding vertices in H. Similarly, for layers N_{k-i} and $N_{k-(i-1)}$. Layers N_i and N_{k-i} are connected using chains in C, each chain which starts in N_i and ends in N_{k-i} will be represented as an edge in H.

   (b) Find a maximum flow between s and t, where the capacity of every edge

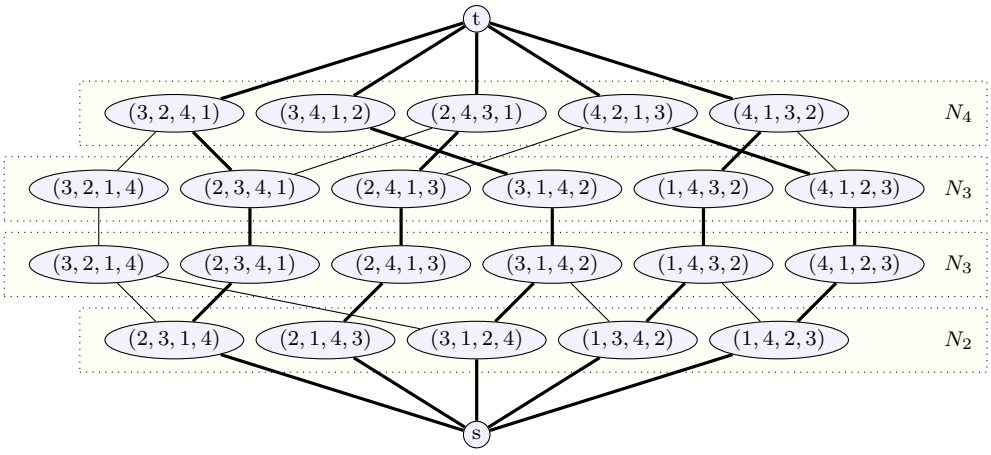
is one. If the value of the flow is equal to $|N_i|$, we remove all edges without any flow, vertices s and t and their adjacent edges. The next step is extending the chains in C, which correspond to the edges between N_i and N_{k-i}, with the flow. The list C contains now a SSCD for subgraph induced by $N_{i-1}, N_i, N_{i+1}, \ldots, N_{k-(i-1)}$ levels of G. If the value is smaller, the poset does not have to have a SSCD.

3. A symmetric saturated chain decomposition is stored in C.

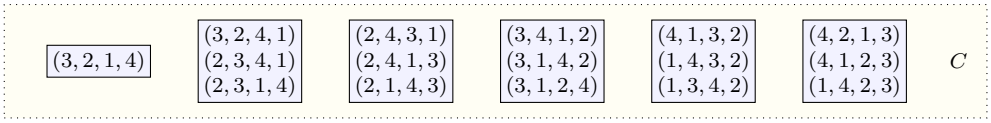
Let us follow steps of this algorithm on I_4. We start by creating the Hasse diagram of the poset and dividing its vertices into seven level sets: $N_0, N_1, \ldots, N_6$ as shown below.



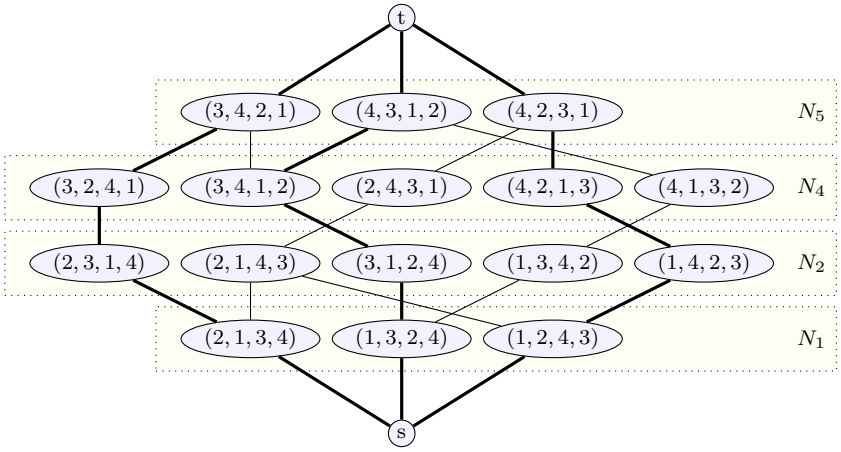
In the next step we save N_3 into C and set $i = 3$ since the rank of I_4 is 6. A graph H contains vertices from levels N_2, N_3, N_3 and N_4 of the original graph. The edges between N_2 and N_3 are determined by the original graph, likewise the edges between N_3 and N_4. Additionally, corresponding vertices in N_3 and N_3 are connected by an edge. The next picture contains a diagram of H with a maximum flow for $i = 3$. The edges with a nonzero flow are denoted by a thicker line.



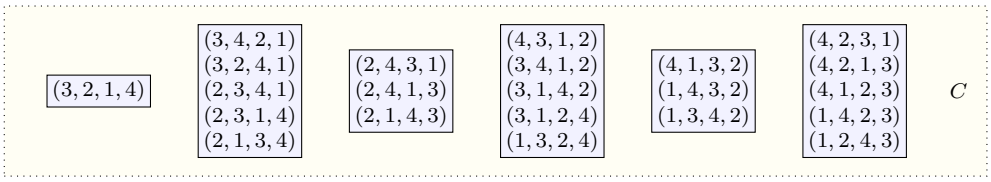
When we find a maximum flow, we remove all edges without any flow and vertices s and t. Remaining chains are used for extending chains in C. The list C is shown beneath.



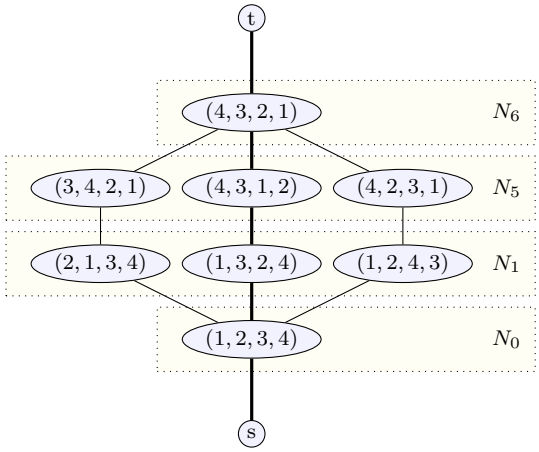
In the next iteration, when $i = 2$, the graph H will contain levels N_1, N_2, N_4 and N_5. Additional vertices s and t are connected to N_1 and N_5, respectively. The diagram of H with a maximum flow is shown below.



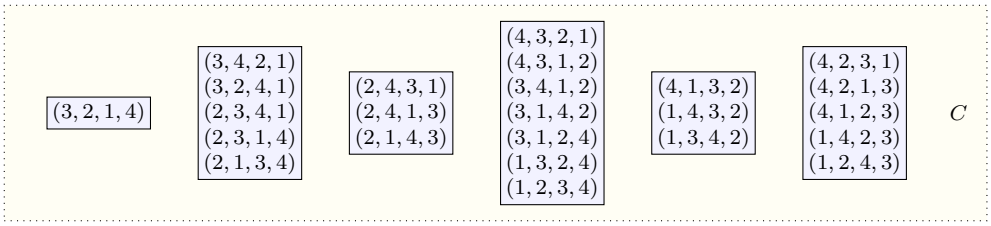
Removing unsaturated edges and vertices s and t from H yields three saturated chains of length 3. These three chains are used to extend some of the existing chains in C. The list C after the extension is illustrated below.



In the last iteration, the graph H contains levels N_0, N_1, N_5 and N_6 since $i = 1$.



A maximum flow in G yields only one chain because N_0 and N_6 contain only one vertex. Therefore, we expand only one chain in C, this is the longest chain in I_4.



When the last chain is extended, C will contain a SSCD of I_4.

The running time of this algorithm is polynomial, since it does not include backtracking. For this reason, the algorithm might not find a SSCD even if a SSCD exists.

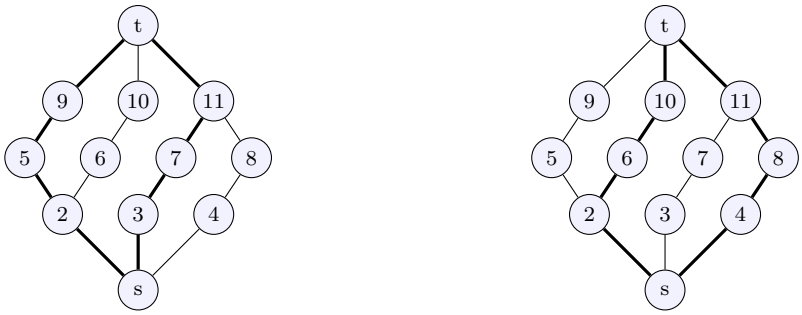


Figure 4.1: Two different flows in a graph.

If the algorithm finds a maximal flow, with its value smaller than the number of neighbors of s or t, it fails to find a SSCD. The algorithm has a running time of $O(n^3m)$ or $O(n^2m^2)$, using the Ford–Fulkerson algorithm, depending on whether a breadth–first or depth–first search is used. The running time is $O(n^2m\log(n^2/m))$ if the Push–Relabel algorithm is used.

Anomalies

Some posets do not have a SSCD. If we run the described algorithm on these posets we should be able to recognize this. One of the first steps of the algorithm, if we run it on the poset shown in Figure 1.1(d), is constructing the graph in Figure 4.1 and finding a maximal flow. Two maximal flows are highlighted in this picture and as we can see, there is always one edge adjacent to s or t with no flow. The algorithm continues with removing s, t and edges without any flow, this step will continue without any problems. But contracting all remaining edges will not create symmetric saturated chains and the algorithm fails. Actually, the algorithm fails on this poset every time; there is at least one edge adjacent to s or t with no flow and our implementations of this algorithm contains such a check. This behavior is needed, since there is no SSCD of this poset.

As mentioned before, the described algorithm does not include backtracking. The algorithm processes a poset from middle layer(s) outward and it does not return to already processed layers. This is the main reason why the running time is polynomial,

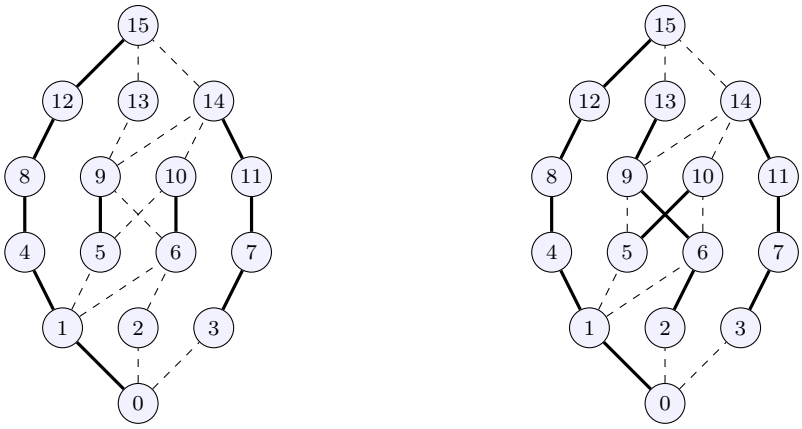


Figure 4.2: An example of two different maximal flows between middle layers and how they can affect the poset decomposition.

but on the other hand this may cause additional problems. While a SSCD contains a maximal flow between layers which are symmetric to the middle layer(s), not every maximal flow between these layers is extendable to a SSCD. There are two choices for a maximal flow between the middle layers for the poset shown in Figure 4.2. If the algorithm picks the maximal flow which contains the edge $(5, 9)$, then it fails to find a SSCD because there will be always two unconnected vertices as in Figure 4.2(a). On the other hand if we choose the second maximal flow between middle layers, we would be able to extend this chains into a SSCD, this is shown in Figure 4.2(b).

There is no simple solution for this problem, the algorithm would have to contain backtracking to solve this and the total running time of such an algorithm would be significantly larger. Fortunately, we were able to find a SSCD for all important posets we ran this algorithm on, even with this imperfection. This was surprising and led us to additional statistics described in the next chapter focused on maximal chains in a poset.

Chapter 5

Results

Two computer programs, which are listed in appendix C, were written using Python and C++. Both of these programs use advanced graph libraries – NetworkX and PyGraphviz in Python and Boost libraries in C++. Both programs can find a SSCD for an inversion poset given its multiset or for a Young's lattice given its two constants m and n. The program written in C++ can additionally find a SSCD for a Boolean algebra and it can read a poset from files. To make sure that our programs provide correct results we checked some decompositions by another program. This additional program checked that the decomposition contained each element of a posed exactly once and that the chains were valid and symmetric.

While the Boost libraries implement the Push–Relabel algorithm, the NetworkX library in Python does not implement this algorithm. Unfortunately, NetworkX implements only the Ford–Fulkerson algorithm with a depth–first search approach. The running time of this algorithm allowed us to find a SSCD, in a reasonable time, for all inversion posets where the multiset contained less than 11 elements. This was a big improvement from the work of Katsumata but we were able to successfully run this program on an inversion poset with 11 distinct elements after improving the Ford–Fulkerson algorithm.

We observed that the performance of the Ford–Fulkerson algorithm with the depth–first search varies significantly. The algorithm is fast at the beginning but it slows down rapidly. On the other hand, the breadth–first search is slower than the depth–first search for paths at the beginning, but its performance does not change over time. It seems that the best solution can be achieved by combining these two approaches. Our implementation of the Ford–Fulkerson algorithm uses the depth–first search first and it switches to the breadth–first search when the breadth–first search is faster.

Additionally, we managed to improve the breadth–first search as well. The regular breadth–first search from a source in a graph would visit all neighbors of the source first, then it would visit neighbors of those neighbors, etc. Our improved implementation of the breadth–first search randomly chooses some of source's neighbors and performs full breadth–first search on them. If it will not find a path from the source to the sink it continues with the next part of nonvisited neighbors of the source until it finds a path from the source to the sink. We achieved the best performance when the algorithm processed 2% of source's neighbors at a time.

The implementation in C++ does not need this improvement because Boost libraries contain an implementation of the Push–Relabel algorithm. This flow algorithm is faster than our improved Ford–Fulkerson algorithm. Since this program is faster than implementation in Python, we extended the input options for it. It can find a SSCD for inversion posets, Young's lattices, Boolean algebras and it can read a poset from files with its structure.

As mentioned in Chapter 3, we can use Corollary 3.9 when testing all inversion posets. Suppose that we would like to find a SSCD for all inversion posets where the multiset contains 4 distinct elements and its length is 6. Let us denote the elements as $1, 2, 3$ and 4, where the order is naturally $1 < 2 < 3 < 4$. There are 10 different possible multisets. This means that we would have to run the program ten times, but using Corollary 3.9 we can reduce the number of instances to 6 because some of these multisets will create isomorphic posets. A list with these combinations of multisets follows.

Poset name	**Size**
Boolean algebra B_n	$n \leq 25$
Inversion poset for a multiset I_N	$\|N\| \leq 11$
Young's lattice $L(m, n)$	$m + n \leq 30$

Table 5.1: Summary of posets with the maximal size for which a symmetric saturated chain decomposition was found by our program.

1. $\{1, 1, 1, 2, 3, 4\}$, $\{1, 2, 3, 4, 4, 4\}$
2. $\{1, 2, 2, 2, 3, 4\}$, $\{1, 2, 3, 3, 3, 4\}$
3. $\{1, 1, 2, 2, 3, 4\}$, $\{1, 2, 3, 3, 4, 4\}$
4. $\{1, 1, 2, 3, 3, 4\}$, $\{1, 2, 2, 3, 4, 4\}$
5. $\{1, 1, 2, 3, 4, 4\}$
6. $\{1, 2, 2, 3, 3, 4\}$

Each multiset in one row will yield the same poset, up to isomorphism. There are 2047 multisets with less than 12 elements, but only 1086 of them are different up to isomorphism.

As mentioned before, the program written in C++ is faster than the program in Python and therefore all our tests were performed with this program. Table 5.1 contains all posets we tested and for which we found a SSCD. It is surprising that the program never failed and always found a SSCD for all tested inversion posets, Young's lattices and Boolean algebras since we provided a poset for which our program fails. It may be possible to extend these boundaries a little bit, but this depends on the computer which runs the program.

n	Number of different longest chains in $\mathbf{I_n}$	Bad chains
4	$16 = \frac{6!}{5 \cdot 3 \cdot 3}$	4
5	$768 = \frac{10!}{7 \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 3}$	147
6	$292{,}864 = \frac{15!}{9 \cdot 7 \cdot 7 \cdot 5 \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 3 \cdot 3}$	24,363
7	$1{,}100{,}742{,}656 = \frac{21!}{11 \cdot 9^2 \cdot 7^3 \cdot 5^4 \cdot 3^5}$	Out of reach
8	$48{,}608{,}795{,}688{,}960 = \frac{28!}{13 \cdot 11^2 \cdot 9^3 \cdot 7^4 \cdot 5^5 \cdot 3^6}$	Out of reach
9	$29{,}258{,}366{,}996{,}258{,}488{,}320 = \frac{36!}{15 \cdot 13^2 \cdot 11^3 \cdot 9^4 \cdot 7^5 \cdot 5^6 \cdot 3^7}$	Out of reach
10	$273{,}035{,}280{,}663{,}535{,}522{,}487{,}992{,}320 = \frac{45!}{17 \cdot 15^2 \cdot 13^3 \cdot 11^4 \cdot 9^5 \cdot 7^6 \cdot 5^7 \cdot 3^8}$	Out of reach

Table 5.2: Number of different longest chains in inversion posets I_n, where $4 \leq n \leq 10$.

5.1 Future work

There are several ways this work might be extended. From a computational point of view, the main reason we could not run our programs on bigger posets was the amount of memory needed. Our implementations construct a poset and keep it as list of vertices and list of edges in the memory. However, because we have to keep only three layers of the poset in memory, the whole poset can be saved in external memory and loaded into internal memory only when needed. This approach, when using a compression, should be investigated in the future.

The memory issue may have an another solution. Distributing the vertex and edge lists over more computers should allow to improve the upper bounds we set in this paper. Distributed flow algorithms are available, although the required communication can be a new limiting factor.

We modified our program to count the number of maximal chains in inversion posets I_n and to find a SSCD given a maximal chain. The number of maximal chains corresponds to the number of different standard Young tableaux of shape $(n-1, n-2, \ldots, 2, 1)$ and it can be found by the Hook formula [11]. This is another piece of evidence that

our program performs correctly. The program did not find a SSCD for some maximal chains; we call these chains bad chains. Table 5.2 shows how many maximal chains I_n has and for small n, how many bad chains I_n has. For example, in I_4, 4 of the 16 maximal chains cannot be extended to a SSCD by our algorithm. Since our algorithm can fail to find a SSCD, table 5.2 provides only upper bounds on the number of bad chains. However, we checked that the exact number of bad chains for I_4 is 4.

Another way to find a SSCD can be by finding symmetric chains from the longest to the shortest. This approach itself may not perform well, but it may be combined with the approach used in our algorithm, since it seems that the majority of the longest chains will not make the rest of the poset undecomposable into symmetric saturated chains.

From a mathematical point of view, Stanley's conjecture that there is a SSCD for any Young's lattice as well as for any inversion poset is still open. However, some progress has been made in determining the Sperner property and other combinatorial properties of these and similar posets [3].

References

[1] A. Abian, *The theory of sets and transfinite arithmetic*, Saunders mathematics books, Saunders, 1965.

[2] I. Anderson, *Combinatorics of Finite Sets*, Dover books on mathematics, Dover Publications, 1987.

[3] V. Dhand, *Tropical decomposition of youngs partition lattice*, Journal of Algebraic Combinatorics, (2013), pp. 1–24.

[4] K. Engel, *Sperner Theory*, Cambridge Solid State Science Series, Cambridge University Press, 1997.

[5] A. Goldberg, E. Tardos, and R. Tarjan, *Network flow algorithms*, in Paths, Flows and VLSI–Layout, B. Korte, L. Lovász, H. Prömel, and A. Schrijver, eds., Springer, 1990, ch. Network Flow Algorithms, pp. 101–164.

[6] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. ACM, 35 (1988), pp. 921–940.

[7] M. Katsumata, *Spernerity and Symmetric Chain Decompositions for the Symmetric Group*, Southern Illinois University at Carbondale, Department of Mathematics, 1987.

[8] K. Krueger, *Two types of bit string generating functions*, master's thesis, University of Minnesota, Duluth, 2012.

[9] R. Stanley, *Weyl groups, the hard lefschetz theorem, and the sperner property*, SIAM Journal on Algebraic Discrete Methods, 1 (1980), pp. 168–184.

[10] R. Stanley, *Enumerative Combinatorics*, no. v. 1 in Cambridge studies in advanced mathematics, Cambridge University Press, 2002.

[11] D. Stanton and D. White, *Constructive*, Undergraduate Texts in Mathematics Series, Springer-Verlag, 1986.

Appendix A

Program interface

We explain how to compile and run programs we developed and what each input parameter means in this chapter. In our description we will assume that we work on a Debian-based Linux machine.

A.1 Python

To successfully run this program we need to have files *diagram.py* and *main.py* inside a folder and install two additional Python libraries, NetworkX and PyGraphviz. The program can be called from a terminal by using this command

```
$ python main.py [-h] [--inversion 1 2 3 4 ...] [--young m n] [--no_pict]
```

The optional arguments are

- --h or --help
  - This argument prints a help message and exits the program.
- --inversion
  - This argument starts the program for an inversion poset and a name of the minimal vertex with the smallest rank follows as a list of numbers, each

separated by a space. For example a SSCD for I_5 can be found by this command

```
$ python main.py --inversion 1 2 3 4 5
```

- --young
  - Two numbers, m and n, separated by a space follow this argument. The program tries to find a SSCD for a Young's lattice $L(m, n)$. For example a SSCD for $L(5, 5)$ can be found by this command

    ```
    $ python main.py --young 5 5
    ```

- --no_pict
  - The program automatically stores the decomposition in a text file and it creates pictures of the Hasse diagram of the poset with the decomposition highlighted. When using this argument, no pictures will be created. Generating pictures for big posets like I_{10} is time consuming and the usability of such pictures is questionable.

If the program finds a SSCD it saves it together with its output into a text file and creates four pictures, depending on the --no_pict argument.

"..._FULL.png"	Contains all vertices with labels. Each level is ordered exactly as it is saved in the memory.
"..._FULL_NO_LABELS.png"	Contains all vertices without labels. Each level is ordered exactly as it is saved in the memory.
"..._SPARSE.png"	Contains all vertices with labels. Each level is ordered such that the final graph has less intersecting edges.

"..._SPARCE_NO_LABELS.png" Contains all vertices without labels. Each level is ordered such that the final graph has less intersecting edges.

A.2 C++

To be able to compile the program we need to install Boost libraries. Optionally, we can install the OpenMP library if we have more CPUs available. The program can be compiled by this command

```
$ g++ -o main main.cpp node.cpp node.h -lboost_program_options
        -lboost_serialization -lboost_system -lboost_filesystem
        -I/usr/include/python2.7 -lpython2.7 -fopenmp -O3
```

assuming that Python 2.7 is installed on the system. This will generate an executable file named "main" which contains our program. The program can be called by using this command

```
$ ./main [--bool n] [--file name] [--inversion 1 2 3 4 ...] [--young m n]
        [--no_pict] [--help]
```

The program in C++ shares some arguments with the program in Python and their effects are the same. These arguments are --inversion, --young, --no_pict and --help. The rest of arguments are described below.

- --bool
  - This argument together with a number n will find a SSCD for a Boolean algebra B_n. The program call for B_6 is

    ```
    $ ./main --bool 6
    ```

- --file

  A string (name) follows this argument. The current folder has to contain two files, "name_levels" and "name_nodes". As their names may suggest,

"name_levels" contains nodes separated into layers and "name_nodes" contains neighbors of each vertex corresponding to the line number. Vertices have to be identified by sequential numbers starting from zero. For example the poset, for which our algorithm fails, from Figure 4.2 can be saved into two files:

my_levels:

```
0
1,2,3
4,5,6,7
8,9,10,11
12,13,14
15
```

my_nodes:

```
1,2,3
4,5,6
6
7
8
9,10
10,9
11
12
13,14
14
14
15
15
15
15
```

As we see, the last vertex has to be connected to itself. The algorithm will fail to find a SSCD for this configuration, but the poset has a SSCD.

If we change the sixth line of *my_nodes* to "9,10" instead of "10,9" the algorithm will find the SSCD because it will find a maximal flow which is shown in Figure 4.2(b). The program can be called by

```
$ ./main --file my
```

Additionally, all pictures are generated by the Python class *diagram.py* and therefore this class has to be in the same folder as the executed *main* file if the argument --no_pict is not used. If the argument --no_pict is used, then program generate the same pictures as the program written in Python.

Appendix B

Performance of programs and generated figures

We provide running times for our programs in selected posets in Table B.1 and Figures B.1 and B.2. Note that these times do not include the time needed to generate the poset in memory.

Program version	Time needed to find a SSCD					
Inversion poset, I_n						
$n =$	6	7	8	9	10	11
C++	1 s	1 s	1 s	15 s	175 s	38.5 min
Python, Ford–Fulkerson from NetworkX	1 s	5.1 s	278.7 s	11.72 h	n/a	n/a
Python, improved Ford–Fulkerson	1 s	2.5 s	75.2 s	2.86 h	n/a	n/a
Young's lattice, $L(m,n)$						
$(m,n) =$	(7,7)	(8,8)	(9,9)	(10,10)	(11,11)	(12,12)
C++	1 s	1 s	2 s	8 s	35 s	170 s
Python, Ford–Fulkerson from NetworkX	2.1 s	25.6 s	453.4 s	148.9 min	56.3 h	n/a
Python, improved Ford–Fulkerson	1.3 s	12.4 s	141.0 s	37.1 min	12.4 h	n/a

Table B.1: Performance of each program on selected posets.

When we focus on the graphs which correspond to the program written in C++, it appears that the last three points or each graph form a line. Since the axes of both graphs have a logarithmic scale, we can estimate the program complexity using the

slopes of these lines. The slope for the line which approximates the performance for inversion posets is 1.07; this implies that the running time is $O(n!^{1.07})$. Similarly, the slope of the line which approximates the last three points for Young's lattices is 1.14 which corresponds to $O\left(\binom{m+n}{n}^{1.14}\right)$. Using these estimations we can conclude that if we had a computer with enough memory, if the program did not fail, we would need 9.2 and 22.2 hours to find a SSCD for I_{12} and $L(16, 16)$, respectively.

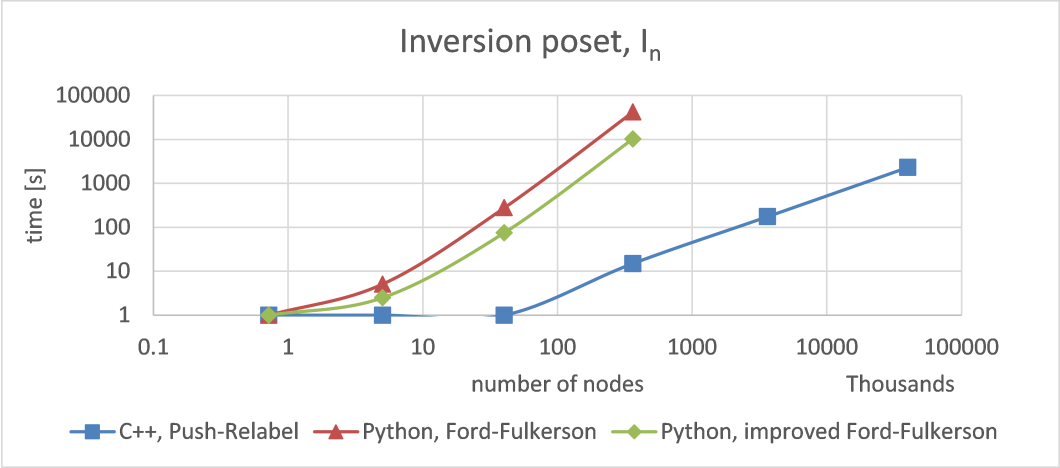


Figure B.1: Performance of programs on inversion posets.

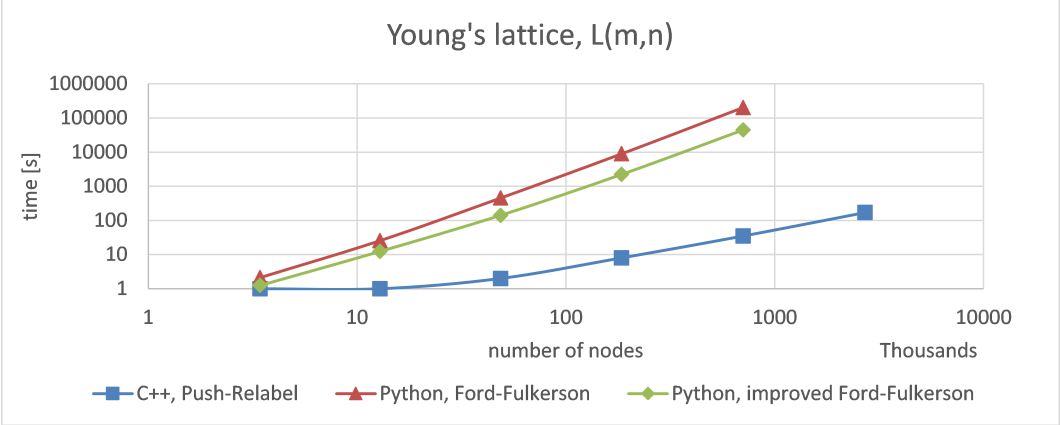
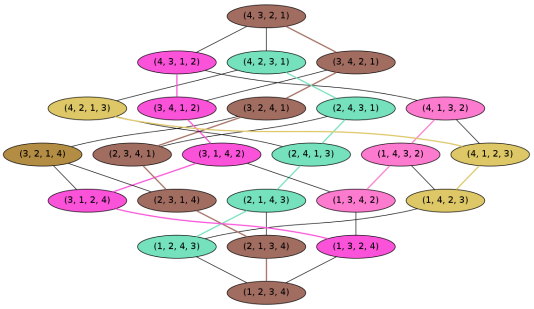
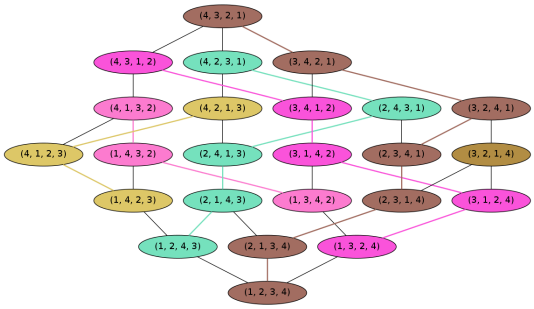


Figure B.2: Performance of programs on Young's lattices.

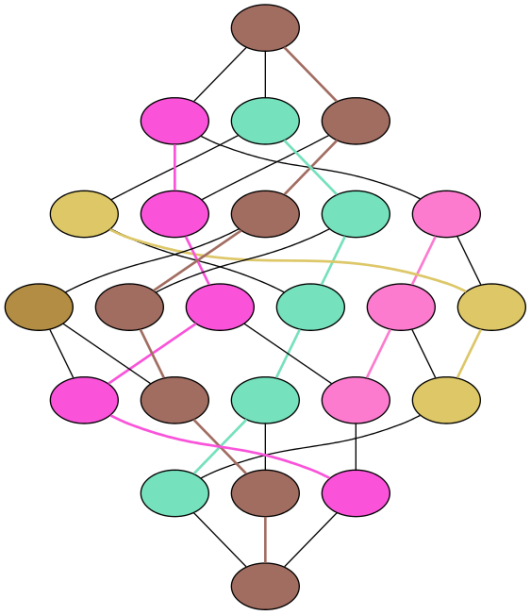
Selected figures generated by the program are displayed below and on the following pages.



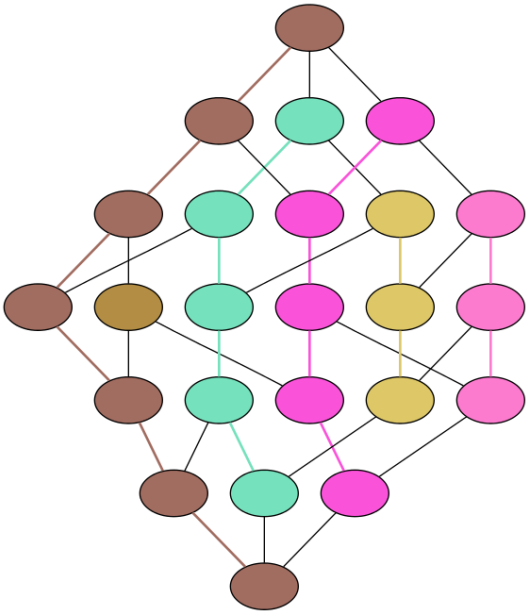
(a) 1,2,3,4_FULL.png



(b) 1,2,3,4_SPARSE.png



(c) 1,2,3,4_FULL_NO_LABELS.png



(d) 1,2,3,4_SPARSE_NO_LABELS.png

Figure B.3: Inversion poset, I_4

Figure B.4: I_5: 1,2,3,4,5_FULL_NO_LABELS.png

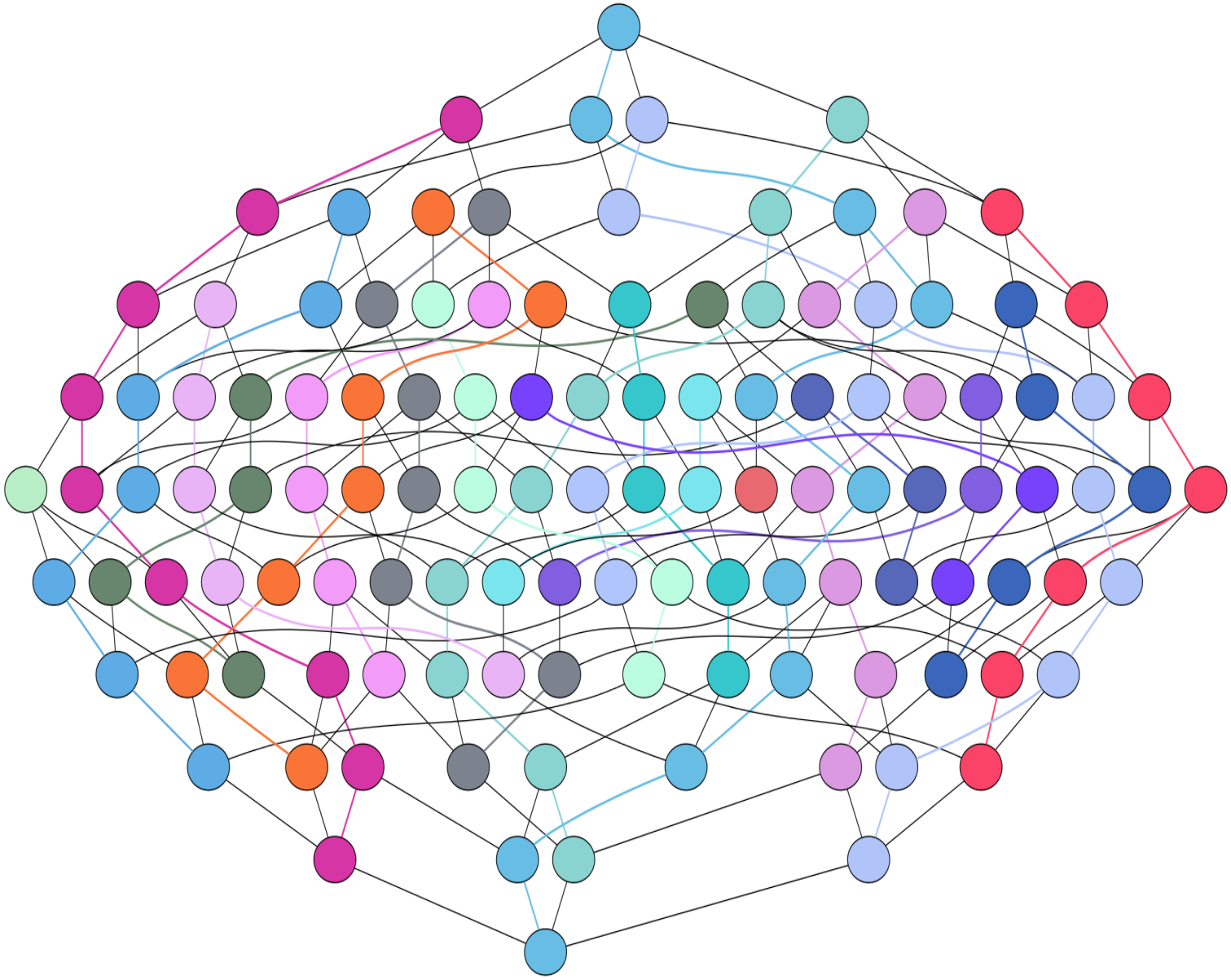


Figure B.5: I_5: 1,2,3,4,5_SPARSE_NO_LABELS.png

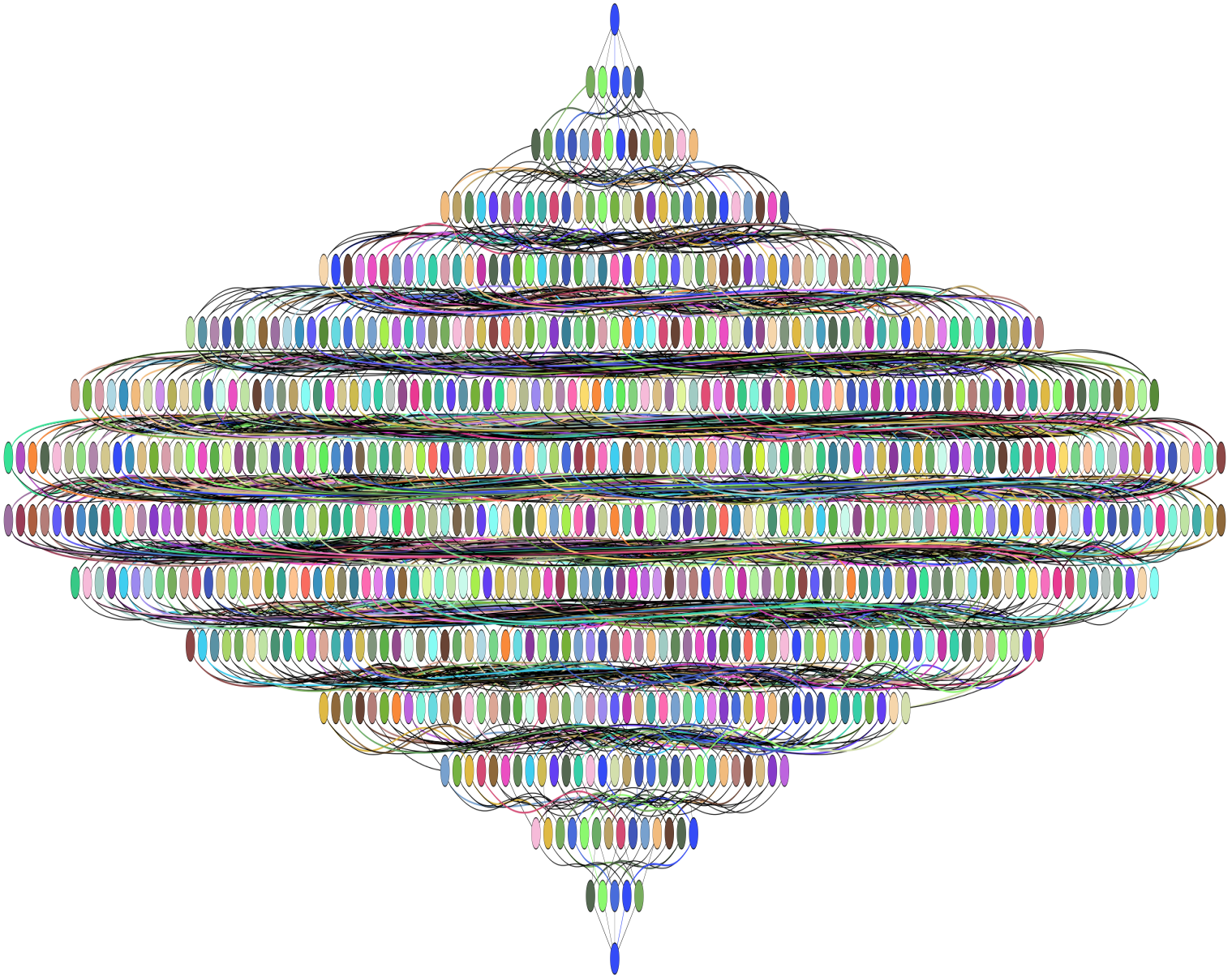


Figure B.6: I_6: 1,2,3,4,5,6_FULL_NO_LABELS.png

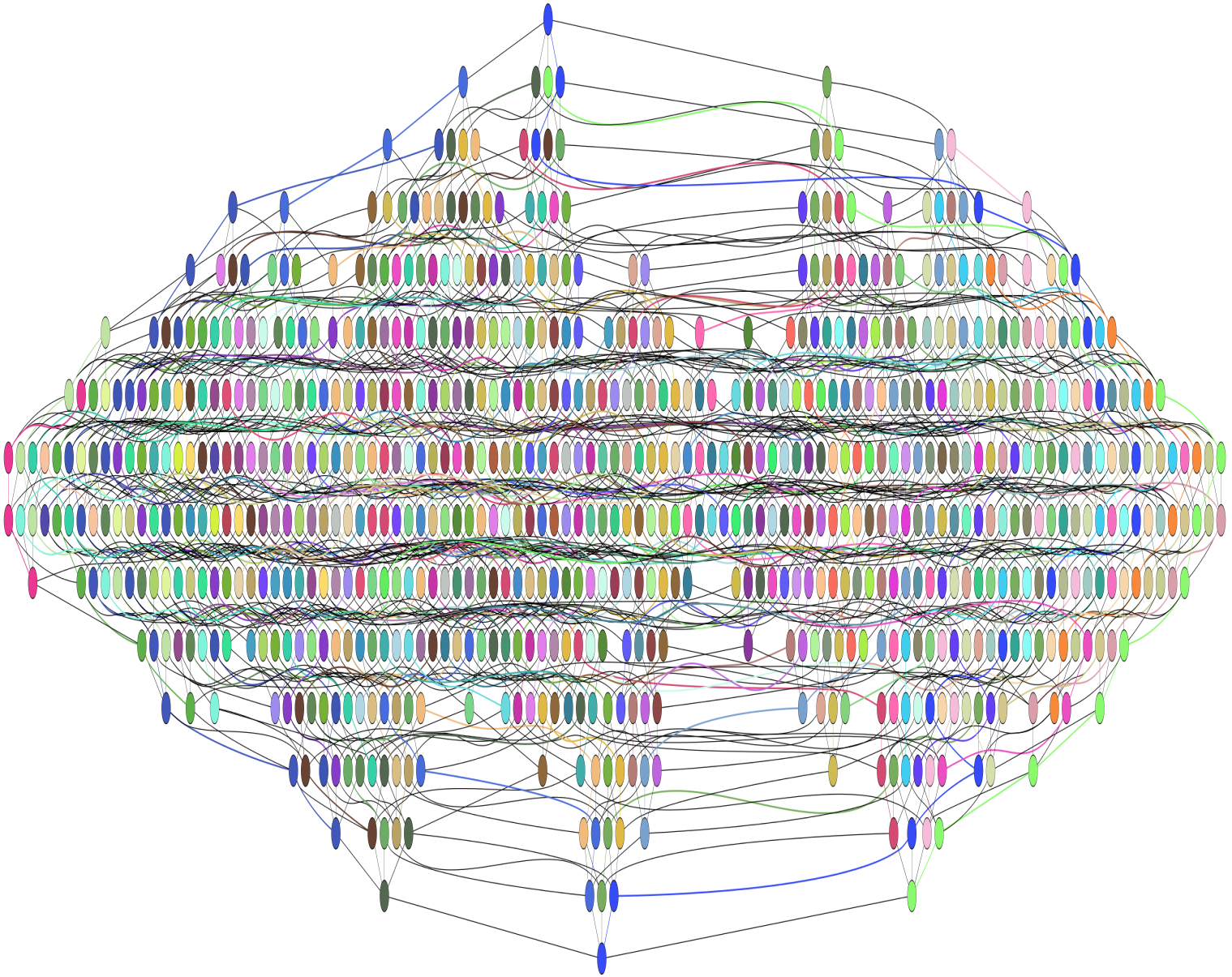
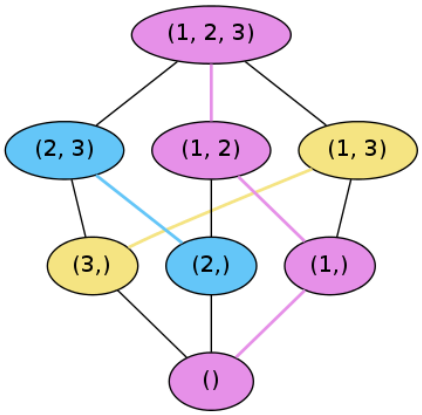
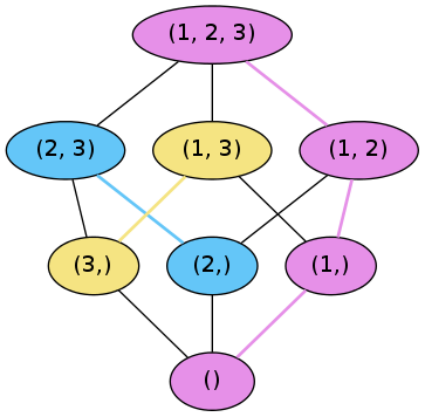


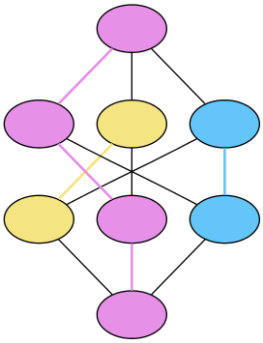
Figure B.7: I_6: 1,2,3,4,5,6_SPARSE_NO_LABELS.png



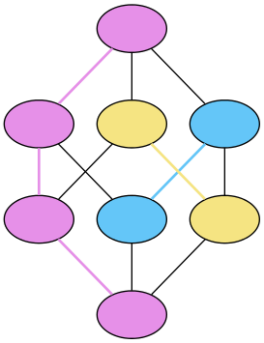
(a) 3_FULL.png



(b) 3_SPARSE.png

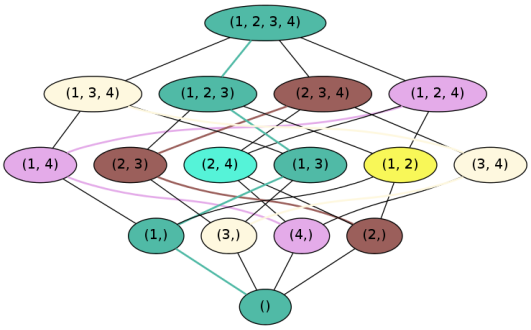


(c) 3_FULL_NO_LABELS.png

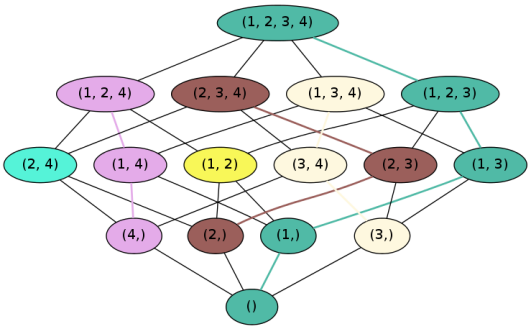


(d) 3_SPARSE_NO_LABELS.png

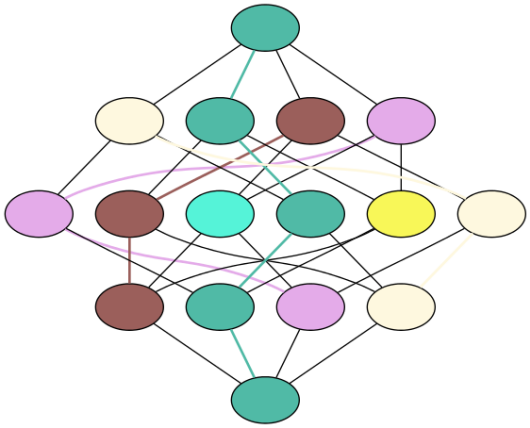
Figure B.8: Boolean algebra, B_3



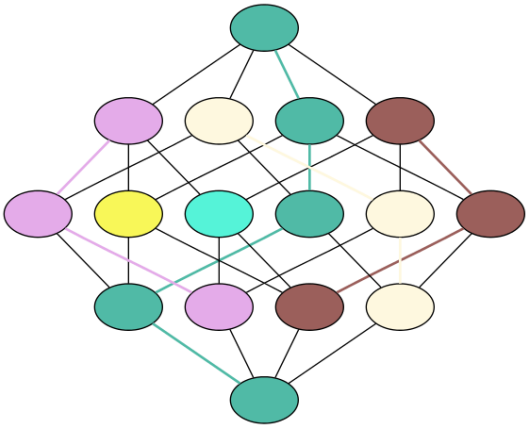
(a) 4_FULL.png



(b) 4_SPARSE.png



(c) 4_FULL_NO_LABELS.png



(d) 4_SPARSE_NO_LABELS.png

Figure B.9: Boolean algebra, B_3

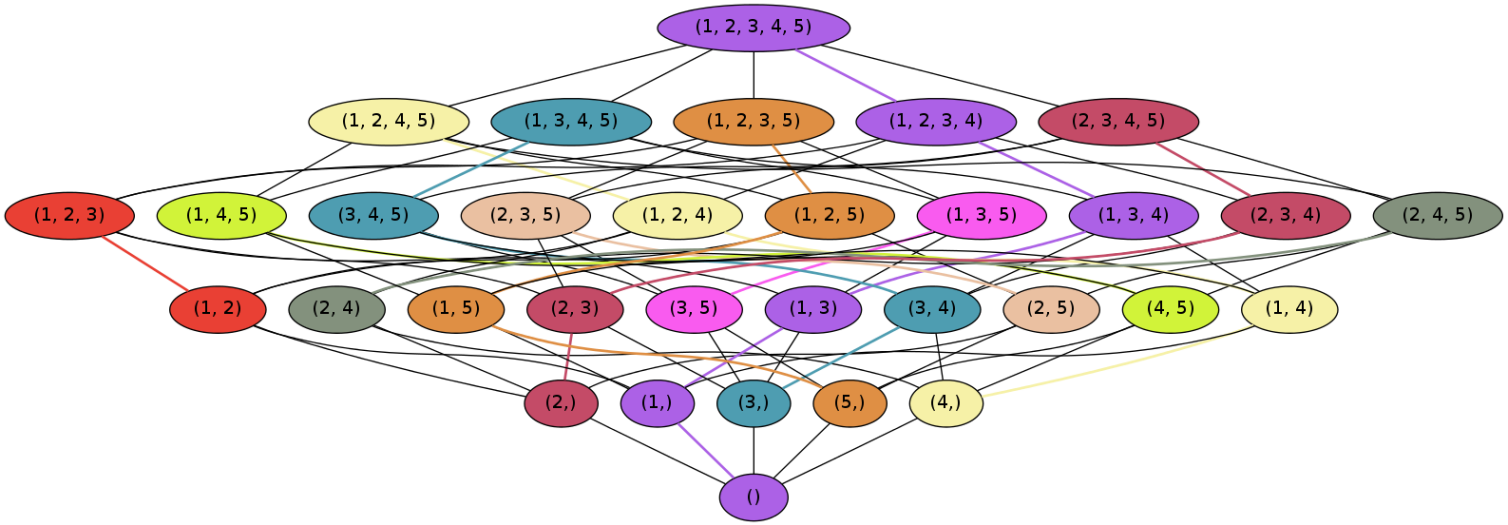


Figure B.10: B_5: 5_FULL.png

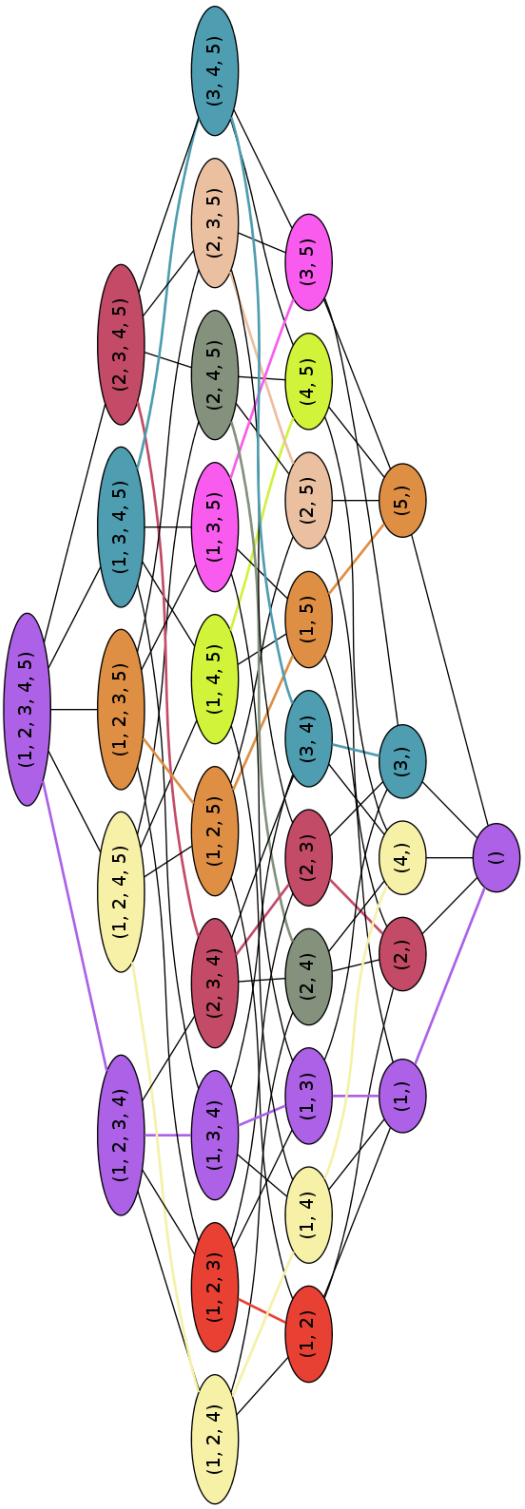
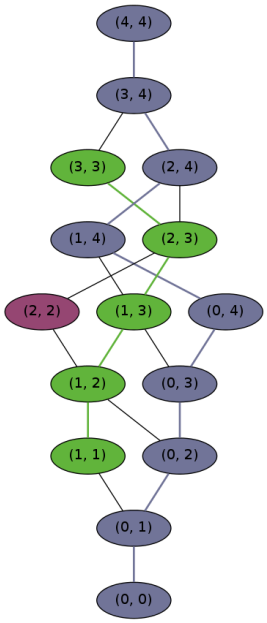
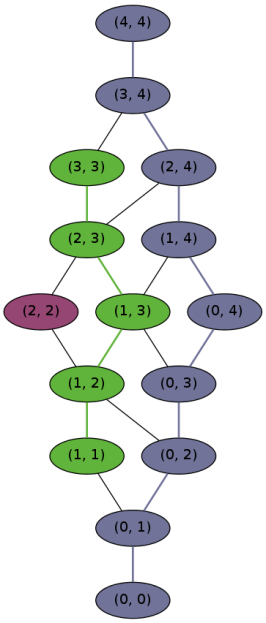


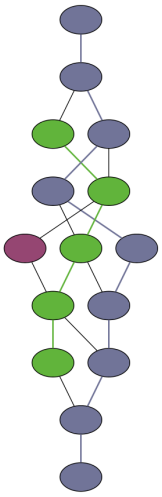
Figure B.11: B_5: 5_SPARSE.png



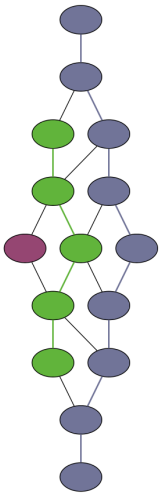
(a) 4,4_FULL.png



(b) 4,4_SPARSE.png

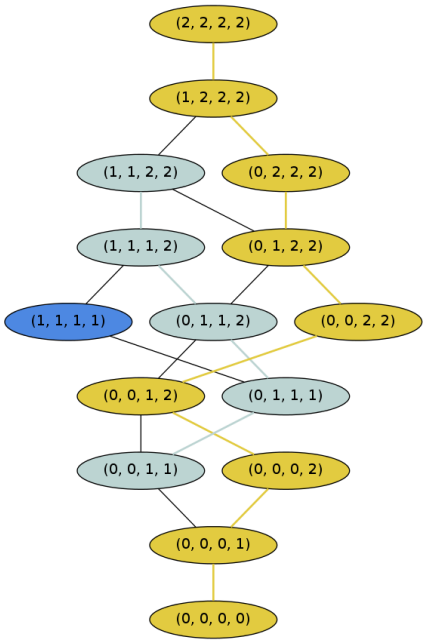


(c) 4,4_FULL_NO_LABELS.png

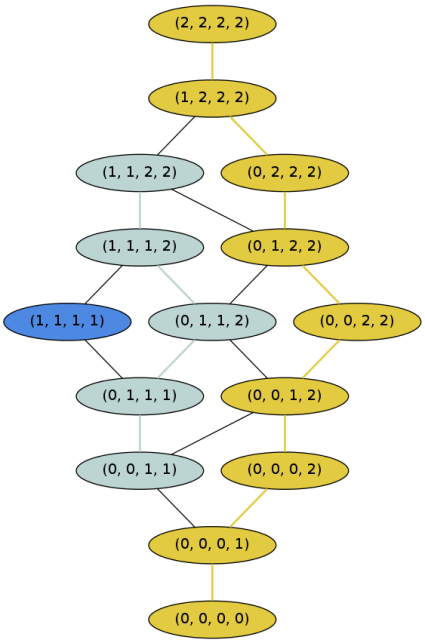


(d) 4,4_SPARSE_NO_LABELS.png

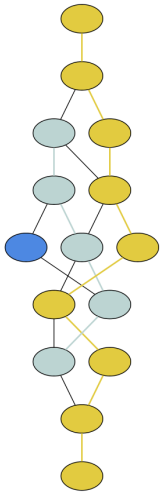
Figure B.12: Young's lattice, $L(4, 2)$



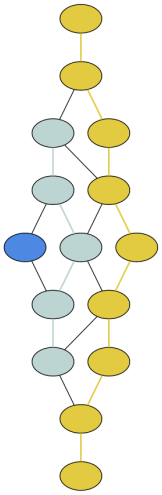
(a) 2,2,2,2_FULL.png



(b) 2,2,2,2_SPARSE.png



(c) 2,2,2,2_FULL_NO_LABELS.png



(d) 2,2,2,2_SPARSE_NO_LABELS.png

Figure B.13: Young's lattice, $L(2,4)$

Appendix C

Source codes

The purpose of this section is to provide two implementations, in Python and in C++, of the algorithm given in the text.

C.1 Python

To run the Python implementation we need two external libraries NetworkX and PyGraphviz. The Ford–Fulkerson algorithm is implemented in the NetworkX library and the PyGraphviz library creates figures of the poset and its symmetric saturated chain decomposition.

The first script tries to find a SSCD. If this script finds a SSCD, it saves the decomposition into a text file and calls the second script (diagram.py) to generate figures of the poset and its decomposition. The first script accompanied with comments is listed below.

main.py:

```
1 #!/usr/bin/python
2
3 ## list of libraries
4 from array import array
5 import networkx as nx
6 import itertools
```

```
7  import time
8  import collections
9  import sys
10 import math
11 import random
12 import argparse
13 import diagram
14
15 ## global variable for the number of already found paths
16 global gCount
17
18 class Node:
19     ## each node has 4 properties
20     ##  name: name of the node
21     ##  level: the number of the level
22     ##  neighbors: list of neighbors in the next level
23     ##  nameOfMatchingNode: name of the matched node
24     def __init__(self, name, level):
25         self.name = name
26         self.level = level
27         self.neighbors = list()
28         self.nameOfMatchingNode = name
29     ## checks if there is possible inversion of its elements
30     def isLast(self):
31         for i in xrange(len(self.name)-1):
32             if self.name[i] < self.name[i+1]:
33                 return False
34         return True
35     ## generate next inversion
36     def generateNext(self,dictOfNextLevel):
37         for i in xrange(len(self.name)-1):
38             if self.name[i] < self.name[i+1]:
39                 tmpName = list(self.name)
40                 tmp = self.name[i]
41                 tmpName[i] = tmpName[i+1]
42                 tmpName[i+1] = tmp
43                 self.neighbors.append(tuple(tmpName))
44                 if ~dictOfNextLevel.has_key(tuple(tmpName)):
45                     dictOfNextLevel[tuple(tmpName)] = Node(tmpName,self.level+1)
46     @property
47     def Name(self):
48         return tuple(self.name)
49     @property
50     def NameOfMatchingNode(self):
51         return tuple(self.nameOfMatchingNode)
52
53 ## logger for easier handeling with output
54 def loggerOut(*message):
55     msg = ""
56     for m in message:
57         msg = ''.join([msg,str(m)])
58     print msg
59     f.write(''.join([msg,'\n']))
60     sys.stdout.flush()
61
62 ## constructing an Inverse poset given a multiset
63 def makeGraphOfInversePartialyOrderedMultiSet(rootList):
64     listOfLevels = [dict(),dict()]
65     root = Node(rootList,0)
```

```
66      listOfLevels[0][tuple(root.name)] = root
67      root.generateNext(listOfLevels[1])
68      level = 1
69      while (len(listOfLevels[level]) > 1) or (not listOfLevels[level].values()[0].
            isLast()):
70          listOfLevels.append(dict())
71          d = listOfLevels[level]
72          for node in d.itervalues():
73              node.generateNext(listOfLevels[level+1])
74          level += 1
75      return listOfLevels
76
77  ## constructing a Young lattice given its size
78  def makeGraphOfYoungsLattice(m,n):
79      listOfLevels = [dict() for x in xrange(m*n+1)]
80      tmp = [0 for i in xrange(n)]
81      root = Node(tmp,0)
82      listOfLevels[0][root.Name] = root
83      for i in xrange(len(listOfLevels)-1):
84          for node in listOfLevels[i].values():
85              if sum(node.Name) < m**n:
86                  for ii in xrange(n):
87                      if (node.Name[ii] < m) and ( (ii == n-1) or (node.Name[ii] <
                            node.Name[ii+1]) ):
88                          new = list(node.name)
89                          new[ii] += 1
90                          newNode = Node(new, i+1)
91                          if listOfLevels[i+1].has_key(newNode.Name) == True:
92                              node.neighbors.append(listOfLevels[i+1][newNode.Name
                                    ].Name)
93                          else:
94                              node.neighbors.append(newNode.Name)
95                              listOfLevels[i+1][newNode.Name] = newNode
96      return listOfLevels
97
98  ## Expanding the graphs for additional layers.
99  def expandGraph(graph,middleLayer,listOfLevels,intActualLayer):
100     if (len(graph) == 1) and (intActualLayer == -1): return
101     graph.clear()
102     for node in listOfLevels[intActualLayer].values():
103         graph.add_edges_from([(node.Name,listOfLevels[intActualLayer+1][neighbor
                ].NameOfMatchingNode) for neighbor in node.neighbors], capacity=1.0)
104     for node in listOfLevels[len(listOfLevels)-intActualLayer-2].values():
105         graph.add_edges_from([(node.NameOfMatchingNode,listOfLevels[len(
                listOfLevels)-intActualLayer-1][neighbor].Name) for neighbor in node.
                neighbors], capacity=1.0)
106
107 ## Adding 'source' and 'sink' nodes for a flow algorithm.
108 def addEndNodes(graph,bottomLevel,upperLevel):
109     graph.add_edges_from([('rootDown',node) for node in bottomLevel], capacity
            =1.0)
110     graph.add_edges_from([(node,'rootUp') for node in upperLevel], capacity=1.0)
111
112 ## Removing edges without any flow.
113 def removeExtraEdges(graph,flow,middleLayer):
114     graph.remove_nodes_from(('rootUp','rootDown'))
115     for node in middleLayer.values():
116         for nodeTmp in graph.successors((node.NameOfMatchingNode,'u')):
117             if flow[(node.NameOfMatchingNode,'u')][nodeTmp] == 0:
```

```
118                     graph.remove_edge((node.NameOfMatchingNode,'u'),nodeTmp)
119             for nodeTmp in graph.predecessors((node.NameOfMatchingNode,'d')):
120                 if flow[nodeTmp][(node.NameOfMatchingNode,'d')] == 0:
121                     graph.remove_edge(nodeTmp,(node.NameOfMatchingNode,'d'))
122
123 ## Contracting edges with a flow - exanding a list with the SSCD.
124 def contractEdges(graph,bottomLayer,middleLayer,upperLayer):
125     graphNew = nx.DiGraph()
126     for node in middleLayer.values():
127         if len(graph.successors((node.NameOfMatchingNode,'u'))) > 0:
128             graph.add_path((graph.predecessors((node.NameOfMatchingNode,'d'))[0],
                    node.NameOfMatchingNode,graph.successors((node.NameOfMatchingNode
                    ,'u'))[0]))
129         else:
130             graph.add_node(node.NameOfMatchingNode)
131         graph.remove_nodes_from(((node.NameOfMatchingNode,'u'),(node.
                NameOfMatchingNode,'d')))
132     for node in bottomLayer.keys():
133         neigh = graph.neighbors(node)[0]
134         graph.remove_node(node)
135         nodeNew = (node,neigh,graph.neighbors(neigh)[0])
136         graph.remove_nodes_from(nodeNew[1:3])
137         graphNew.add_node(nodeNew)
138         bottomLayer[nodeNew[0]].nameOfMatchingNode = nodeNew
139         upperLayer[nodeNew[2]].nameOfMatchingNode = nodeNew
140     return graphNew, graph.nodes()
141
142     result = None
143
144 ## Finding a path between source and sink using depth-first approach.
145 def find_pathDepthFirst(graph, visitedNodes, source, sink, path):
146     result = None
147     visitedNodes.append(source)
148     if source == sink:
149         return path
150     for succesor in graph.successors(source):
151         residual = graph.edge[source][succesor]['capacity'] - graph.edge[source][
                succesor]['flow']
152         if residual > 0 and not succesor in visitedNodes:
153             if not succesor in path:
154                 result = find_pathDepthFirst( graph, visitedNodes, succesor, sink
                        , path + [succesor] )
155             if result != None:
156                 return result
157     for predecessor in graph.predecessors(source):
158         residual = graph.edge[predecessor][source]['flow']
159         if residual > 0 and not predecessor in visitedNodes:
160             if not predecessor in path:
161                 result = find_pathDepthFirst( graph, visitedNodes, predecessor,
                        sink, path + [predecessor] )
162             if result != None:
163                 return result
164
165 ## Finding a path between source and sink using enhanced breadth-first approach.
        This function process nodes attached to the source in blocks. Each block
        contain certain percentage of neighbors to source. The breadt-first search
        then continue with those nodes. If no path to the sink is found, then the
        function investigate next block of neighbors.
166 def find_pathBreadthFirst(graph, sink, actualLayer, layers, percentage):
```

```
167     if percentage == 0:
168         raise Exception('Percentage cannot be 0!')
169     visitedNodes = dict()
170     ## erase visited tokens in graph at the beginning
171     if actualLayer == 0:
172         if not layers.has_key(1):
173             layers[1] = list()
174         for source, path in layers[0]:
175             visitedNodes[source] = True
176             for succesor in graph.successors(source):
177                 residual = graph.edge[source][succesor]['capacity'] - graph.edge[
                        source][succesor]['flow']
178                 if (residual > 0) and (not succesor in path) and (not
                        visitedNodes.has_key(succesor)):
179                     visitedNodes[succesor] = True
180                     layers[1].append([succesor, path+[succesor]])
181         actualLayer += 1
182     ## change from percentage to a number of nodes
183     if (actualLayer == 1) and (percentage <= 1):
184         percentage = math.ceil(len(layers[1]) * percentage)
185 
186     while actualLayer > 0:
187         newNodes = False
188         if not layers.has_key(actualLayer+1):
189             layers[actualLayer+1] = list()
190         ## neighbors of the source node
191         if actualLayer == 1:
192             for i in xrange(int(percentage)):
193                 if len(layers[actualLayer]) == 0:
194                     break
195                 item = layers[actualLayer].pop(random.choice(xrange(len(layers[
                        actualLayer]))))
196                 source = item[0]
197                 path = item[1]
198                 if source == sink:
199                     return path
200                 for succesor in graph.successors(source):
201                     residual = graph.edge[source][succesor]['capacity'] - graph.
                            edge[source][succesor]['flow']
202                     if residual > 0 and (not succesor in path) and (not
                            visitedNodes.has_key(succesor)):
203                         layers[actualLayer+1].append([succesor, path+[succesor]])
204                         visitedNodes[succesor] = True
205                         newNodes = True
206                 for predecessor in graph.predecessors(source):
207                     residual = graph.edge[predecessor][source]['flow']
208                     if residual > 0 and (not predecessor in path) and (not
                            visitedNodes.has_key(predecessor)):
209                         layers[actualLayer+1].append([predecessor, path+[
                                predecessor]])
210                         visitedNodes[predecessor] = True
211                         newNodes = True
212 ## Investigate the whole path for nodes not attached to source.
213         else:
214             for source, path in layers[actualLayer]:
215                 if source == sink:
216                     return path
217                 for succesor in graph.successors(source):
```

```
218                     residual = graph.edge[source][succesor]['capacity'] - graph.
                            edge[source][succesor]['flow']
219                     if residual > 0 and (not succesor in path) and (not
                            visitedNodes.has_key(succesor)):
220                         layers[actualLayer+1].append([succesor, path+[succesor]])
221                         visitedNodes[succesor] = True
222                         newNodes = True
223                 for predecessor in graph.predecessors(source):
224                     residual = graph.edge[predecessor][source]['flow']
225                     if residual > 0 and (not predecessor in path) and (not
                            visitedNodes.has_key(predecessor)):
226                         layers[actualLayer+1].append([predecessor, path+[
                                predecessor]])
227                         visitedNodes[predecessor] = True
228                         newNodes = True
229         if newNodes:
230             actualLayer += 1
231         else:
232             actualLayer -= 1
233 
234     return None
235 
236 ## Ford-Fulkerson algorithm with adaptive switching between depth and breadth
        search
237 def myFord_fulkerson(graph, source, sink):
238     for x in graph.edge:
239         for y in graph.edge[x]:
240             graph.edge[x][y]['flow'] = 0
241     path = find_pathDepthFirst(graph, [], source, sink, [source])
242     tmpTime = time.time()
243     tmp = find_pathBreadthFirst(graph, sink, 0 , {0:[[source,[source]]]}, .02)
244     timeBreadth = time.time() - tmpTime
245     value = 0
246     breadth = False
247     while path != None:
248         value += 1
249         for i in xrange(1,len(path)):
250             if graph.edge[path[i-1]].has_key(path[i]):
251                 graph.edge[path[i-1]][path[i]]['flow'] = 1
252             else:
253                 graph.edge[path[i]][path[i-1]]['flow'] = 0
254         if breadth:
255             path = find_pathBreadthFirst(graph, sink, 0 , {0:[[source,[source
                    ]]]}, .02)
256         else:
257             start = time.time()
258             path = find_pathDepthFirst(graph, [], source, sink, [source])
259             if time.time() - start > timeBreadth*2:
260                 breadth = True
261                 loggerOut('--_--_Breadth-search_activated!_--_--')
262     flow = dict()
263     for x,y in graph.edges():
264         f = graph.edge[x][y].pop('flow')
265         if flow.has_key(x):
266             flow[x][y] = f
267         else:
268             flow[x] = {y: f}
269     return value, flow
270 
```

```
271 ## Extending the middle layer and finding a maximal flow through a graph.
272 def extendAndFlow(graph,middleLayer,bottomLayer):
273     for node in middleLayer.values():
274         neighDown = graph.predecessors(node.NameOfMatchingNode)
275         neighUp = graph.successors(node.NameOfMatchingNode)
276         graph.remove_node(node.NameOfMatchingNode)
277         graph.add_edge((node.NameOfMatchingNode,'d'),(node.NameOfMatchingNode,'u'
                ), capacity=1.0)
278         graph.add_edges_from([((node.NameOfMatchingNode,'u'),x) for x in neighUp
                ], capacity=1.0)
279         graph.add_edges_from([(x,(node.NameOfMatchingNode,'d')) for x in
                neighDown], capacity=1.0)
280     value, flow = myFord_fulkerson(graph,'rootDown','rootUp')
281     return value, flow
282
283 ## Main body of the program.
284 def main(f, listOfLevels):
285     gCount = 0
286     start = time.time()
287     numberOfLayers = len(listOfLevels)-1
288     decomposition = list()
289     graph = nx.DiGraph()
290     middleLayer = list()
291     if numberOfLayers%2 == 0:
292   ## there is one layer in the middle
293   loggerOut('--␣One␣layer␣in␣the␣middle␣--')
294   intActualLayer = numberOfLayers/2-1
295   ## Make a graph with three layers.
296   loggerOut('--␣Middle␣part␣start␣--')
297   for i in xrange(numberOfLayers/2-1,numberOfLayers/2+1):
298       for item in listOfLevels[i].values():
299     for iitem in item.neighbors:
300         graph.add_edge(item.Name,iitem, capacity=1.0)
301   for item in listOfLevels[numberOfLayers/2].values():
302       middleLayer.append(item.Name)
303   loggerOut('--␣Middle␣part␣done␣--',(time.time() - start), "seconds.")
304     else:
305   ## There are two middle layers -> find matching, create a graph with 3 layers
306   loggerOut('--␣Two␣layers␣in␣the␣middle␣--')
307   intActualLayer = numberOfLayers/2
308   for item in listOfLevels[intActualLayer].values():
309       for itemNeighbor in item.neighbors:
310     tmpNode = listOfLevels[intActualLayer+1].get(itemNeighbor)
311     graph.add_edge(item.Name, tmpNode.Name)
312   tmp = nx.DiGraph()
313   loggerOut('--␣matching␣start␣--',(time.time() - start), "seconds.")
314   matching = nx.max_weight_matching(graph)
315   loggerOut('--␣matching␣end␣--',(time.time() - start), "seconds.")
316   ## Contract the matching into symmetric chains.
317   for item in matching.keys():
318       if listOfLevels[intActualLayer].has_key(item):
319     newNode = (item,matching[item])
320     listOfLevels[intActualLayer].get(item).nameOfMatchingNode = newNode
321     listOfLevels[intActualLayer+1].get(matching[item]).nameOfMatchingNode =
            newNode
322     middleLayer.append(newNode)
323   intActualLayer -= 1
324   tmp.add_nodes_from(middleLayer)
325   graph = tmp
```

```
326   ## Add bottom neighbors for a matching.
327   for item in listOfLevels[intActualLayer].values():
328       for itemNeighbor in item.neighbors:
329     tmpNode = listOfLevels[intActualLayer+1].get(itemNeighbor).NameOfMatchingNode
330     graph.add_edge(item.Name, tmpNode)
331   ## Add upper neighbors for a matching.
332   for item in listOfLevels[numberOfLayers-intActualLayer-1].values():
333       for itemNeighbor in item.neighbors:
334     tmpNode = listOfLevels[numberOfLayers-intActualLayer].get(itemNeighbor).Name
335     graph.add_edge(item.NameOfMatchingNode,tmpNode)
336
337     while (intActualLayer >= 0):
338   loggerOut('-- ActualLayer is ',intActualLayer,'/',numberOfLayers,' --',(time.
          time() - start), "seconds.")
339   addEndNodes(graph,listOfLevels[intActualLayer].keys(),listOfLevels[
          numberOfLayers-intActualLayer].keys())
340   loggerOut('-- flow start --',(time.time() - start), "seconds.")
341   value, flow = extendAndFlow(graph,listOfLevels[intActualLayer+1],listOfLevels[
          intActualLayer].keys())
342   if value < len(listOfLevels[intActualLayer]):
343       raise Exception('NO matching')
344   loggerOut('-- flow end --',(time.time() - start), "seconds.")
345   removeExtraEdges(graph,flow,listOfLevels[intActualLayer+1])
346   loggerOut('-- removeExtraEdges end --',(time.time() - start), "seconds.")
347   graph, disconnectedNodes = contractEdges(graph,listOfLevels[intActualLayer],
          listOfLevels[intActualLayer+1],listOfLevels[numberOfLayers-intActualLayer])
348   decomposition.extend(disconnectedNodes)
349   intActualLayer -= 1
350   loggerOut('-- expandGraph start --',(time.time() - start), "seconds.")
351   expandGraph(graph,middleLayer,listOfLevels,intActualLayer)
352   loggerOut('-- expandGraph done --',(time.time() - start), "seconds.")
353     decomposition.extend(graph.nodes())
354     end = time.time()
355     elapsed = end - start
356     loggerOut("Time: ", (time.time() - start), "seconds.")
357     loggerOut('\nDecomposition:')
358     for i in decomposition:
359   if type(i[0]) == tuple:
360       msg = ""
361       for m in i:
362     msg = ''.join([msg,str(m)])
363   else:
364       msg = str(i)
365   f.write(''.join([msg,'\n']))
366     f.close()
367
368 ## Processing user input, running the main function and generating graphs
369 parser = argparse.ArgumentParser(description='Program for finding a symmetric
          saturated chain decomposition, created as a part of Master Thesis by Ondrej
          Zjevik at University of Minnesota.',usage='%(prog)s [-h] [--inversion 1 2 3 4
          ...] [--young m n]')
370 parser.add_argument('--inversion', type=int, nargs='+', help='element with rank 0
          ', metavar='1 2 3 4 ...', dest='root')
371 parser.add_argument('--young', type=int, nargs=2, help="constants of a Young's
          lattice", metavar=('m','n'))
372 parser.add_argument('--no_pict', dest='picture', default=True, const=False,
          action='store_const', metavar='', help="will not generate pictures")
373 args = parser.parse_args()
374 if(args.root == None and args.young == None):
```

```
    parser.print_help()
    exit()
sys.setrecursionlimit(10000)
if(args.root != None):
    listOfLevels = makeGraphOfInversePartialyOrderedMultiSet(args.root)
    f = open(''.join(['','.join(map(str, args.root)),'.txt']),'w')
    main(f, listOfLevels)
    if args.picture:
  diagram.makeGraphOfInversePartialyOrderedMultiSet(f,args.root)
else:
    listOfLevels = makeGraphOfYoungsLattice(args.young[0], args.young[1])
    f = open(''.join(['','.join(map(str, [args.young[0] for i in xrange(args.young
        [1])]))],'.txt']),'w')
    main(f, listOfLevels)
    if args.picture:
  diagram.makeGraphOfYoungsLattice(f,args.young[0], args.young[1])
```

The second script written in Python generates pictures from an output from the first script or from the script written in C++. The content of the second script follows.

diagram.py:

```
#!/usr/bin/python

from array import array
import pygraphviz as pgv
from random import randrange
color = True;

class Node:
    def __init__(self, name, level):
        self.name = name
        self.level = level
        self.neighbors = list()
        self.nameOfMatchingNode = name
    def isLast(self):
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                return False
        return True
    def generateNext(self,dictOfNextLevel,listOfNextLevel,graph):
        graph.add_node(self.Name)
        for i in xrange(len(self.name)-1):
            if self.name[i] < self.name[i+1]:
                tmpName = list(self.name)
                tmp = self.name[i]
                tmpName[i] = tmpName[i+1]
                tmpName[i+1] = tmp
                self.neighbors.append(tuple(tmpName))
                graph.add_edge(self.Name,tuple(tmpName),weight=1,style='solid')
                if ~dictOfNextLevel.has_key(tuple(tmpName)):
                    tmpNode = Node(tmpName,self.level+1)
                    dictOfNextLevel[tuple(tmpName)] = tmpNode
                    listOfNextLevel.append(tmpNode)
    def generateNextBool(self,dictOfNextLevel,listOfNextLevel,graph,size):
        graph.add_node(self.Name)
```

```
35          for i in xrange(1,size+1):
36              if self.name.count(i) == 0:
37                  tmpName = list(self.name)
38                  tmpName.append(i)
39                  tmpName.sort()
40                  self.neighbors.append(tuple(tmpName))
41                  graph.add_edge(self.Name,tuple(tmpName),weight=1,style='solid')
42                  if ~dictOfNextLevel.has_key(tuple(tmpName)):
43                      tmpNode = Node(tmpName,self.level+1)
44                      dictOfNextLevel[tuple(tmpName)] = tmpNode
45                      listOfNextLevel.append(tmpNode)
46      @property
47      def Name(self):
48          return tuple(self.name)
49      @property
50      def NameOfMatchingNode(self):
51          return tuple(self.nameOfMatchingNode)
52
53  def makeBool(graph, size):
54      dictOfLevels = [dict(),dict()]
55      listOfLevels = [list(),list()]
56      root = Node(tuple(),0)
57      dictOfLevels[0][tuple(root.name)] = root
58      listOfLevels[0].append(root)
59      root.generateNextBool(dictOfLevels[1],listOfLevels[1],graph,size)
60      level = 1
61      while (len(dictOfLevels[level]) > 1):
62          dictOfLevels.append(dict())
63          listOfLevels.append(list())
64          d = dictOfLevels[level]
65          for node in d.itervalues():
66              node.generateNextBool(dictOfLevels[level+1],listOfLevels[level+1],
                    graph,size)
67          level += 1
68      sparseGraph = graph.copy()
69      for level in xrange(1,len(listOfLevels)):
70          for node in dictOfLevels[level-1].keys():
71              for node2 in dictOfLevels[level].keys():
72                  if graph.has_edge(node2,node) == False:
73                      graph.add_edge(node,node2,weight=1,style='invis')
74      return sparseGraph
75
76  def makeInversePartialyOrderedMultiSet(graph, root):
77      dictOfLevels = [dict(),dict()]
78      listOfLevels = [list(),list()]
79      root = Node(root,0)
80      dictOfLevels[0][tuple(root.name)] = root
81      listOfLevels[0].append(root)
82      root.generateNext(dictOfLevels[1],listOfLevels[1],graph)
83      level = 1
84      while (len(dictOfLevels[level]) > 1) or (not dictOfLevels[level].values()[0].
            isLast()):
85          dictOfLevels.append(dict())
86          listOfLevels.append(list())
87          d = dictOfLevels[level]
88          for node in d.itervalues():
89              node.generateNext(dictOfLevels[level+1],listOfLevels[level+1],graph)
90          level += 1
91      sparseGraph = graph.copy()
```

```
92      for level in xrange(1,len(listOfLevels)):
93          for node in dictOfLevels[level-1].keys():
94              for node2 in dictOfLevels[level].keys():
95                  if graph.has_edge(node2,node) == False:
96                      graph.add_edge(node,node2,weight=1,style='invis')
97      return sparseGraph
98
99  def makeYoungsLattice(graph, m, n):
100     dictOfLevels = [dict() for x in xrange(m*n+1)]
101     tmp = [0 for i in xrange(n)]
102     root = Node(tmp,0)
103     dictOfLevels[0][root.Name] = root
104     for i in xrange(len(dictOfLevels)-1):
105         for node in dictOfLevels[i].values():
106             if sum(node.Name) <= m*n:
107                 for ii in xrange(n):
108                     if (node.Name[ii] < m) and ( (ii == n-1) or (node.Name[ii] <
                            node.Name[ii+1]) ):
109                         new = list(node.name)
110                         new[ii] += 1
111                         newNode = Node(new, i+1)
112                         graph.add_edge(newNode.Name,node.Name,weight=1,style='
                                solid')
113                         if dictOfLevels[i+1].has_key(newNode.Name) == True:
114                             node.neighbors.append(dictOfLevels[i+1][newNode.Name
                                    ].Name)
115                         else:
116                             node.neighbors.append(newNode.Name)
117                             dictOfLevels[i+1][newNode.Name] = newNode
118     sparseGraph = graph.copy()
119     for level in xrange(1,m*n):
120         for node in dictOfLevels[level-1].keys():
121             for node2 in dictOfLevels[level].keys():
122                 if graph.has_edge(node2,node) == False:
123                     graph.add_edge(node2,node,weight=1,style='invis')
124     return sparseGraph
125
126 def makeFromFile(graph, fileName):
127     listOfLevels = []
128     f = open(fileName+"_levels",'r')
129     level = -1;
130     for line in f:
131   listOfLevels.append(list())
132   level += 1;
133   line.strip()
134   for node in line.split(','):
135       listOfLevels[level].append(int(node))
136
137     f = open(fileName+"_nodes",'r')
138     nodeId = -1;
139     for line in f:
140   nodeId += 1;
141   if nodeId > level: continue
142   line.strip()
143   for neighbor in line.split(','):
144       graph.add_edge((int(nodeId),),(int(neighbor),),weight=1,style='solid')
145
146     sparseGraph = graph.copy()
147     for level in xrange(1,len(listOfLevels)):
```

```
148         for node in listOfLevels[level-1]:
149             for node2 in listOfLevels[level]:
150                 if graph.has_edge((node2,),(node,)) == False:
151                     graph.add_edge((node,),(node2,),weight=1,style='invis')
152     return sparseGraph
153 
154 def makeGraphOfBool(oldFile, size):
155     graph = pgv.AGraph(rankdir='BT')
156     sparseGraph = makeBool(graph,size)
157 
158     decomp = False;
159     i = 0
160     d = dict()
161     f = open(oldFile.name,'r')
162     generatePictures(f,graph,sparseGraph)
163 
164 def makeGraphOfInversePartialyOrderedMultiSet(oldFile, root):
165     graph = pgv.AGraph(rankdir='BT')
166     sparseGraph = makeInversePartialyOrderedMultiSet(graph,root)
167 
168     decomp = False;
169     i = 0
170     d = dict()
171     f = open(oldFile.name,'r')
172     generatePictures(f,graph,sparseGraph)
173 
174 def makeGraphOfYoungsLattice(oldFile,m,n):
175     f = open(oldFile.name,'r')
176     graph = pgv.AGraph()
177     numberOfLayers = m*n
178     sparseGraph = makeYoungsLattice(graph,m,n)
179     generatePictures(f,graph,sparseGraph)
180 
181 def makeGraphFromFile(decomposition, fileName):
182     graph = pgv.AGraph(rankdir='BT')
183     sparseGraph = makeFromFile(graph,fileName)
184     f = open(decomposition.name,'r')
185     generatePictures(f,graph,sparseGraph)
186 
187 def generatePictures(f,graph,sparseGraph):
188     decomp = False;
189     i = 0
190     d = dict()
191     for line in f:
192   i += 1
193   line.strip()
194 
195   if decomp:
196       line = str.replace(line,'((','(')
197       line = str.replace(line,'),␣','')
198       line = str.replace(line,'))','')
199       line = str.replace(line,')','')
200       line = str.replace(line,'\n','')
201       if str.count(line,',␣') == 0:
202     line = str.replace(line,',',',␣')
203       lineList = list()
204       for part in line.split('(')[1:]:
205     if len(part) > 0:
206         lineList.append(tuple(map(int, part.split(',␣'))))
```

```
207         else:
208             lineList.append(tuple())
209           c = "#%s" % "".join([hex(randrange(50, 255))[2:] for i in range(3)])
210           if(color): graph.node_attr['style'] = 'filled'
211           graph.get_node(lineList[0]).attr['fillcolor'] = c
212           if(color): sparseGraph.node_attr['style'] = 'filled'
213           sparseGraph.get_node(lineList[0]).attr['fillcolor'] = c
214           for ii in xrange(1,len(lineList)):
215         graph.get_node(lineList[ii]).attr['fillcolor'] = c
216         if (color):
217             graph.add_edge(lineList[ii-1],lineList[ii],weight=1,color=c,penwidth=2)
218         else:
219             graph.add_edge(lineList[ii-1],lineList[ii],weight=1)
220         sparseGraph.get_node(lineList[ii]).attr['fillcolor'] = c
221         if(color):
222             sparseGraph.add_edge(lineList[ii-1],lineList[ii],weight=1,color=c,
                    penwidth=2)
223         else:
224             sparseGraph.add_edge(lineList[ii-1],lineList[ii],weight=1)
225
226       if (line == 'Decomposition:\n'): decomp = True
227
228         graphNoLabels = graph.copy()
229         graphNoLabels.node_attr['label'] = '␣'
230         sparseGraphNoLabels = sparseGraph.copy()
231         sparseGraphNoLabels .node_attr['label'] = '␣'
232         graph.layout(prog='dot')
233         graph.draw(str.replace(f.name,'.txt','')+"_FULL.png")
234         sparseGraph.layout(prog='dot')
235         sparseGraph.draw(str.replace(f.name,'.txt','')+"_SPARSE.png")
236         graphNoLabels.layout(prog='dot')
237         graphNoLabels.draw(str.replace(f.name,'.txt','')+"_FULL_NO_LABELS.png")
238         sparseGraphNoLabels .layout(prog='dot')
239         sparseGraphNoLabels.draw(str.replace(f.name,'.txt','')+"_SPARSE_NO_LABELS.png
            ")
```

C.2 C++

We present the source code for the C++language in this section. The program contains two classes, *main* and *node*. Each instance of the node class represents one vertex of the poset and each instance stores an id, vertex label, level index and a list of its neighbors. The classes are given below.

node.h:

```
1
2   #ifndef NODE_H
3   #define NODE_H
4
```

```
5  #include <vector>
6  #include <string>
7  #include <map>
8  #include <fstream>
9  #include <boost/serialization/vector.hpp>
10 #include <boost/serialization/set.hpp>
11 #include <boost/serialization/array.hpp>
12 #include <boost/archive/text_oarchive.hpp>
13 #include <boost/archive/text_iarchive.hpp>
14 #include <boost/shared_ptr.hpp>
15 #include <boost/make_shared.hpp>
16
17 using namespace std;
18
19 typedef unsigned int IdType;
20 typedef unsigned char NameElementType;
21
22 class Node
23 {
24 public:
25   unsigned char level; // Good for up to 255 levels
26   IdType id; // Can store up to 12! elements
27   IdType boolSize; // store the number of elements in a set
28
29   vector<NameElementType> name;
30   vector<IdType> neighbors;
31
32
33   Node();
34   Node(std::vector<NameElementType> name, int level, IdType id);
35   ~Node();
36   void generateNextBool( map<vector<NameElementType>, boost::shared_ptr<Node> >*
         tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
         nextLevel, std::ofstream* fileWithNodeNames);
37   void generateNextInverse( map<vector<NameElementType>, boost::shared_ptr<Node>
         >* tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
         nextLevel, std::ofstream* fileWithNodeNames);
38   void generateNextYoung( map<vector<NameElementType>, boost::shared_ptr<Node> >*
          tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
         nextLevel, std::ofstream* fileWithNodeNames, int m);
39   string getName();
40   string getNameYoung();
41 };
42
43
44
45 #endif // NODE_H
```

node.cpp:

```
1  // libraries:
2  #include "node.h"
3  #include <vector>
4  #include <string>
5  #include <boost/concept_check.hpp>
6  #include <boost/lexical_cast.hpp>
7  using namespace std;
8
9  Node::Node(vector<NameElementType> name, int level, IdType id){
10   this->name = name;
11   this->level = level;
12   this->id = id;
13 }
14 // Default constructors:
15 Node::Node(){
16 }
17
18 Node::~Node(){
19
20 }
21 // generate next vertices in the boolean lattice
22 void Node::generateNextBool( map<vector<NameElementType>, boost::shared_ptr<Node>
       >* tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
     nextLevel, std::ofstream* fileWithNodeNames)
23 {
24   for (NameElementType i = 1; i <= this->boolSize; i++)
25   {
26   if (name.size() == 0 || find(name.begin(),name.end(),i) == name.end() )
27   {
28     vector<NameElementType> tmp;
29     for(int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
30     tmp.push_back(i);
31     sort(tmp.begin(), tmp.end());
32       if(tmp_level->count(tmp) == 0){
33         boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp,level+1,
               adjacentMatrix->size());
34         newNode->boolSize = this->boolSize;
35         string name = newNode->getName();
36         int stringSize = (boolSize <= 9 ? 2+boolSize*2 : 2+9*2+(boolSize-9)*3);
37         (*fileWithNodeNames) << name;
38         for(int k = 0; k < stringSize-name.length(); k++)(*fileWithNodeNames) <<
               "_";
39         (*fileWithNodeNames) << endl;
40         fileWithNodeNames->flush();
41         adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
42         nextLevel->push_back(adjacentMatrix->size());
43         adjacentMatrix->push_back(new vector<IdType>());
44         (*tmp_level)[tmp] = newNode;
45       } else{
46         adjacentMatrix->at(id)->push_back(tmp_level->at(tmp)->id);
47       }
48     }
49   }
50 }
51 // generate next vertices using only inversion on an instantiation
```

```
52 void Node::generateNextInverse( map<vector<NameElementType>, boost::shared_ptr<
       Node> >* tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
        nextLevel, std::ofstream* fileWithNodeNames)
53 {
54   for (unsigned int i = 0; i < name.size()-1; i++)
55   {
56     if (name.at(i) < name.at(i+1))
57     {
58       vector<NameElementType> tmp;
59       for(int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
60       NameElementType t = tmp.at(i);
61       tmp.at(i) = tmp.at(i+1);
62       tmp.at(i+1) = t;
63       if(tmp_level->count(tmp) == 0){
64         boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp,level+1,
               adjacentMatrix->size());
65         string name = newNode->getName();
66         (*fileWithNodeNames) << name << endl;
67         fileWithNodeNames->flush();
68         adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
69         nextLevel->push_back(adjacentMatrix->size());
70         adjacentMatrix->push_back(new vector<IdType>());
71         (*tmp_level)[tmp] = newNode;
72       } else{
73         adjacentMatrix->at(id)->push_back(tmp_level->at(tmp)->id);
74       }
75     }
76   }
77 }
78 // generate next vertices in a Young's lattice
79 void Node::generateNextYoung( map<vector<NameElementType>, boost::shared_ptr<Node
       > >* tmp_level, vector<vector<IdType>* >* adjacentMatrix, vector<IdType>*
       nextLevel, std::ofstream* fileWithNodeNames, int m)
80 {
81   for (int i = 0; i < name.size(); i++)
82   {
83     if (name.at(i) < m && (i == 0 || name[i-1] > name[i]) )
84     {
85       vector<NameElementType> tmp;
86       for(int j = 0; j < name.size(); j++) tmp.push_back(name[j]);
87       tmp.at(i)++;
88       if(tmp_level->count(tmp) == 0){
89         boost::shared_ptr<Node> newNode = boost::make_shared<Node>(tmp,level+1,
               adjacentMatrix->size());
90         string name = newNode->getNameYoung();
91         (*fileWithNodeNames) << name << endl;
92         fileWithNodeNames->flush();
93         adjacentMatrix->at(id)->push_back(adjacentMatrix->size());
94         nextLevel->push_back(adjacentMatrix->size());
95         adjacentMatrix->push_back(new vector<IdType>());
96         (*tmp_level)[tmp] = newNode;
97       } else{
98         adjacentMatrix->at(id)->push_back(tmp_level->at(tmp)->id);
99       }
100    }
101  }
102 }
103 // return name as a string
104 string Node::getName()
```

```
105 {
106   string ret = "(";
107   for(int i = 0; i < (int)name.size()-1; i++){
108     ret += boost::lexical_cast<string>((int)name.at(i));
109     ret += ",";
110   }
111   if(name.size() > 0) ret += boost::lexical_cast<string>((int)name.at(name.size()
          -1));
112   ret += ")";
113   return ret;
114 }
115 // return name as a string
116 string Node::getNameYoung()
117 {
118   string ret = "(";
119   for(unsigned int i = name.size()-1; i > 0; i--){
120     if((int)name.at(i) < 10) ret += "0";
121     ret += boost::lexical_cast<string>((int)name.at(i));
122     ret += ",";
123   }
124   if((int)name.at(0) < 10) ret += "0";
125   ret += boost::lexical_cast<string>((int)name.at(0));
126   ret += ")";
127   return ret;
128 }
```

main.cpp:

```
1  //libraries:
2  #include "algorithm"
3  #include <iostream>
4  #include <fstream>
5  #include <limits>
6  #include <time.h>
7  #include <iterator>
8  #include <vector>
9  #include <map>
10 #include <boost/program_options.hpp>
11 #include <boost/program_options/options_description.hpp>
12 #include <boost/algorithm/string.hpp>
13 #include <boost/graph/adjacency_list.hpp>
14 #include <boost/graph/graphviz.hpp>
15 #include <boost/property_map/property_map.hpp>
16 #include <boost/graph/boykov_kolmogorov_max_flow.hpp>
17 #include <boost/graph/edmonds_karp_max_flow.hpp>
18 #include <boost/graph/push_relabel_max_flow.hpp>
19 #include <boost/graph/properties.hpp>
20 #include <boost/graph/max_cardinality_matching.hpp>
21
22 #include <boost/serialization/map.hpp>
23 #include <boost/serialization/list.hpp>
24 #include <boost/serialization/vector.hpp>
25 #include <boost/serialization/shared_ptr.hpp>
26 #include <boost/archive/text_oarchive.hpp>
27 #include <boost/archive/text_iarchive.hpp>
28 #include <boost/archive/binary_oarchive.hpp>
29 #include <boost/archive/binary_iarchive.hpp>
30 #include "boost/graph/graph_traits.hpp"
31
32 #include "node.h"
33 #include <Python.h>
34
35 using std::vector;
36 using namespace boost;
37 namespace po = boost::program_options;
38
39 typedef adjacency_list_traits < vecS, vecS, directedS > Traits;
40
41 // vertex structure
42 struct Vertex{
43   IdType id;
44   unsigned char level;
45   long distance;
46   default_color_type color;
47   Traits::edge_descriptor predecessor;
48 };
49 // edge structure
50 struct Edge{
51   long capacity;
52   long residual_capacity;
53   Traits::edge_descriptor reverse;
54 };
55
56 typedef adjacency_list < vecS, vecS, directedS, Vertex, Edge> Graph;
57
```

```
58  typedef Graph::vertex_descriptor NodeId;
59  typedef Graph::edge_descriptor EdgeId;
60  Graph graph;
61
62  vector<vector<IdType>* > listOfLevels;
63  time_t start_time;
64  // contains ids of neighbors of a node
65  vector<vector<IdType>* > arrayOfEdges;
66
67  void removeUnusedEdges(Graph *graph, property_map<Graph, long Edge::*>::type*
        m_e_c, property_map<Graph, long Edge::*>::type* m_e_r_c, int bottom_layer,
        int upper_layer);
68  // output file
69  static std::ofstream* output;
70  // logger
71  static void loggerOut(string message){
72    cout << message << "\n";
73    *output << message << "\n";
74  }
75  unsigned int charInLine = 0;
76  // helper class
77  std::istream& GotoLine(std::istream& file, unsigned int num){
78    if(charInLine == 0){
79      file.seekg(std::ios::beg);
80      string line;
81      getline(file,line);
82      charInLine = file.tellg();
83    }
84    file.seekg(charInLine*num);
85    return file;
86  }
87  // add an edge between two vertices and set its parameters
88  EdgeId* AddEdge(Graph::vertex_descriptor &v1, Graph::vertex_descriptor &v2, const
        int capacity, Graph* g)
89  {
90    EdgeId e1 = add_edge(v1, v2, *g).first;
91    EdgeId e2 = add_edge(v2, v1, *g).first;
92    g->operator[](e1).capacity = 1;
93    g->operator[](e1).reverse = e2;
94    g->operator[](e2).capacity = 0;
95    g->operator[](e2).reverse = e1;
96    g->operator[](v2).predecessor = e2;
97    return &e1;
98  }
99  // Uses Push-Relabel flow algorith to find a matching between middle layers.
100 void findMatching(Graph* graph, int layer)
101 {
102   loggerOut("-- Middle part start --");
103   Graph g = *graph;
104   int l = listOfLevels.size()-1 - layer;
105   NodeId sink = add_vertex(*graph);
106   graph->operator[](sink).id = -1;
107   for (vector<IdType>::const_iterator it = listOfLevels[l]->begin(), e =
          listOfLevels[l]->end(); it != e; ++it)
108   {
109     //add the upper layer
110     NodeId n = add_vertex(*graph);  //(*it) - (*listOfLevels[l]->begin()) + 1;
          //id of a node in the graph
111     graph->operator[](n).id = *it;
```

```
112       AddEdge(n,sink,1,graph);
113     }
114     l--;
115
116     NodeId src = add_vertex(*graph);
117     for (vector<IdType>::const_iterator it = listOfLevels[l]->begin(), e =
            listOfLevels[l]->end(); it != e; ++it)
118     {
119       //add bottom layer and connect nodes with its neighbors
120       add_vertex(*graph);
121       NodeId n = (*it) - (*listOfLevels[l]->begin()) + 2 + listOfLevels[l+1]->size
              (); //id of a node in the graph
122       graph->operator[](n).id = *it;
123       AddEdge(src,n,1,graph); //connect with source
124
125       for(vector<IdType>::const_iterator neigbour = arrayOfEdges[*it]->begin(), ee
              = arrayOfEdges[*it]->end(); neigbour != ee; ++neigbour)
126       {
127         NodeId nn = (*neigbour) - (*listOfLevels[l+1]->begin()) + 1;
128         AddEdge(n,nn,1,graph);
129       }
130
131     }
132     property_map<Graph, long Edge::*>::type map_edge_capacity(get(&Edge::capacity,*
            graph));
133     property_map<Graph, long Edge::*>::type map_edge_residual_capacity(get(&Edge::
            residual_capacity,*graph));
134     property_map<Graph, Graph::edge_descriptor Edge::*>::type map_edge_reverse(get
            (&Edge::reverse,*graph));
135     property_map < Graph, vertex_index_t >::type map_vertex_index = get(
            vertex_index,*graph);
136
137     long flow = 0;
138     time_t time_start, time_end;
139     double time_push;
140     time(&time_start);
141     flow = push_relabel_max_flow(*graph,src,sink,map_edge_capacity,
            map_edge_residual_capacity,map_edge_reverse,map_vertex_index);
142     time(&time_end);
143     time_push = difftime(time_end,time_start);
144     time(&time_start);
145
146     loggerOut("--␣Middle␣part␣done␣--"+lexical_cast<string>(time_push));
147     removeUnusedEdges(graph,&map_edge_capacity,&map_edge_residual_capacity, l, l+1)
            ;
148 }
149 // removes edges without any flow
150 void removeUnusedEdges(Graph *graph, property_map<Graph, long Edge::*>::type*
        m_e_c, property_map<Graph, long Edge::*>::type* m_e_r_c, int bottom_layer,
        int top_layer){
151     property_map<Graph, long Edge::*>::type map_edge_capacity = *m_e_c;
152     property_map<Graph, long Edge::*>::type map_edge_residual_capacity = *m_e_r_c;
153     graph_traits<Graph>::vertex_iterator u_iter, u_end,u_it;
154     graph_traits<Graph>::edge_iterator ei, e_end;
155     int id_bottom_first = listOfLevels[bottom_layer]->at(0), id_bottom_last =
            listOfLevels[bottom_layer]->at(listOfLevels[bottom_layer]->size()-1);
156     int id_top_first = listOfLevels[top_layer]->at(0), id_top_last = listOfLevels[
            top_layer]->at(listOfLevels[top_layer]->size()-1);
```

```
157   int id_before_top_first = listOfLevels[top_layer-1]->at(0), id_before_top_last
          = listOfLevels[top_layer-1]->at(listOfLevels[top_layer-1]->size()-1);
158
159   for(int i = id_bottom_first; i <= id_bottom_last; i++) arrayOfEdges[i]->clear()
          ; // erase all links for bottom layer
160   for(int i = id_before_top_first; i <= id_before_top_last; i++) arrayOfEdges[i
          ]->clear(); // erase all links to upper layer
161
162   for(boost::tie(ei,e_end) = edges(*graph); ei != e_end; ei++){
163     if ( map_edge_capacity[*ei] == 1 && (map_edge_capacity[*ei] -
            map_edge_residual_capacity[*ei]) == 1 ){
164       //keep the edge - save the connected node into arrayOfEdges as the only
              neighbor
165       IdType id_source = graph->operator[](source(*ei, *graph)).id;
166       IdType id_target = graph->operator[](target(*ei, *graph)).id;
167       if(id_source >= id_bottom_first && id_source <= id_bottom_last && id_target
            != -1 && id_target != id_source){
168         //edge goes from bottom layer
169         arrayOfEdges[id_source]->push_back(id_target);
170       } else
171       if(id_target >= id_top_first && id_target <= id_top_last && id_target != -1
            && id_target != id_source){
172         //edge goes to the top layer
173         arrayOfEdges[id_source]->push_back(id_target);
174       }
175     }
176   }
177 }
178 // extends the middle layer and run the flow algorithm on the constructed graph
179 int extendAndFlow(Graph* graph, int layer) //layer is the # of actual layer and
        it's going to zero
180 {
181   loggerOut("--␣flow␣start␣--");
182   int upperLayer = listOfLevels.size()-1 - layer;
183   NodeId sink = add_vertex(*graph);
184   graph->operator[](sink).id = -1;
185   NodeId src = add_vertex(*graph);  //source node
186   graph->operator[](src).id = -1;
187
188   //upper layer
189   for (vector<IdType>::const_iterator it = listOfLevels[upperLayer]->begin(), e =
          listOfLevels[upperLayer]->end(); it != e; ++it)
190   {
191     NodeId n = add_vertex(*graph);
192     graph->operator[](n).id = *it;
193     AddEdge(n,sink,1,graph);
194   }
195
196   //middle layer
197   upperLayer--;
198   for (vector<IdType>::const_iterator it = listOfLevels[upperLayer]->begin(), e =
          listOfLevels[upperLayer]->end(); it != e; ++it)
199   {
200     NodeId n = add_vertex(*graph);
201     graph->operator[](n).id = *it;
202
203     NodeId nn = add_vertex(*graph);
204     graph->operator[](nn).id = *it;
205     AddEdge(n,nn,1,graph);  //edge in the middle
```

```
206     for(vector<IdType>::const_iterator neighbour = arrayOfEdges[*it]->begin(), ee
            = arrayOfEdges[*it]->end(); neighbour != ee; ++neighbour)
207     {
208       NodeId tmp = vertex((*neighbour) - (*listOfLevels[upperLayer+1]->begin()) +
              2,*graph);
209       AddEdge(nn,tmp,1,graph);
210     }
211   }
212
213   //bottom layer
214   for (vector<IdType>::const_iterator it = listOfLevels[layer]->begin(), e =
          listOfLevels[layer]->end(); it != e; ++it)
215   {
216     NodeId n = add_vertex(*graph);
217     graph->operator[](n).id = *it;
218     AddEdge(src,n,1,graph);
219
220     for(vector<IdType>::const_iterator neighbor = arrayOfEdges[*it]->begin(), ee
            = arrayOfEdges[*it]->end(); neighbor != ee; ++neighbor){
221       NodeId nn = *neighbor;
222
223       for(int i = layer; i < upperLayer-1; i++) nn = arrayOfEdges[nn]->at(0);
224       nn = vertex(2*(nn - listOfLevels[upperLayer]->at(0))+2+listOfLevels[
              upperLayer+1]->size(),*graph);
225       AddEdge(n,nn,1,graph);
226       graph->operator[](nn).id = *neighbor;
227     }
228   }
229
230
231   property_map<Graph, long Edge::*>::type map_edge_capacity(get(&Edge::capacity,*
          graph));
232   property_map<Graph, long Edge::*>::type map_edge_residual_capacity(get(&Edge::
          residual_capacity,*graph));
233   property_map<Graph, Graph::edge_descriptor Edge::*>::type map_edge_reverse(get
          (&Edge::reverse,*graph));
234   property_map < Graph, vertex_index_t >::type map_vertex_index = get(
          vertex_index,*graph);
235
236   long flow = 0;
237   time_t time_start, time_end;
238   double time_push;
239   time(&time_start);
240   flow = push_relabel_max_flow(*graph,src,sink,map_edge_capacity,
          map_edge_residual_capacity,map_edge_reverse,map_vertex_index);
241   time(&time_end);
242   time_push = difftime(time_end,time_start);
243   time(&time_start);
244
245   loggerOut("--_flow_end_--"+lexical_cast<string>(time_push));
246
247   removeUnusedEdges(graph,&map_edge_capacity,&map_edge_residual_capacity,layer,
          upperLayer+1);
248   return flow;
249 }
250 // the main class, which tries to find a Symmetric Saturated Chain Decomposition
        of poset saved in listOfLevels and in arrayOfEdges
251 void findSSCD(Graph *g, boost::program_options::variables_map vm){
252   Graph graph = *g;
```

```
253   int layer;
254   string name = "nodes.dat";
255
256   if (listOfLevels.size() % 2 == 0)
257   {
258     //Two layers in the middle
259     loggerOut("--Two layers in the middle--");
260     //layer is the number of the actual layer; layer goes to zero
261     layer = listOfLevels.size()/2 - 1;
262     graph.clear();
263     findMatching(&graph, layer--);
264
265   }
266   else
267   {
268     //One layer in the middle
269     loggerOut("--One layer in the middle--");
270     layer = listOfLevels.size()/2 - 1;
271     graph.clear();
272     extendAndFlow(&graph,layer--);
273   }
274   while(layer >= 0){
275     graph.clear();
276     loggerOut("Actual layer is "+lexical_cast<string>(layer)+"/"+lexical_cast<
          string>(listOfLevels.size()));
277     if (extendAndFlow(&graph,layer--) != listOfLevels[layer+1]->size()){
278       loggerOut("--Possibly no symmetric chain decomposition! Terminating...--");
279       output->flush();
280       exit(EXIT_FAILURE);
281     }
282   }
283
284
285   //Print founded symmetric chains decomposition
286   std::ifstream fileWithNodeNameRead(name.c_str());
287   std::istream& nodeNames = fileWithNodeNameRead;
288   name = "decomposition.dat_tmp";
289   std::ofstream fileWithTmpDecomposition(name.c_str(), ios::trunc);
290   string line;
291
292   time_t tmp;
293   time(&tmp);
294   loggerOut("--Total time: "+lexical_cast<string>(tmp-start_time)+"--");
295
296   loggerOut("Decomposition:");
297   if(listOfLevels.size() % 2 == 1){
298     for (vector<IdType>::const_iterator it = listOfLevels[listOfLevels.size()
          /2]->begin(), e = listOfLevels[listOfLevels.size()/2]->end(); it != e; ++
          it){
299       if(arrayOfEdges[*it]->size() == 0){
300         GotoLine(nodeNames, *it);
301         getline(nodeNames,line);
302         trim(line);
303         (vm.count("file")) ? (fileWithTmpDecomposition << "(" << *it << ")" <<
              endl) : (fileWithTmpDecomposition << line << endl);
304       }
305     }
306   }
307   for (int i = 0; i < listOfLevels.size()/2; i++){
```

```
308     for (vector<IdType>::const_iterator it = listOfLevels[i]->begin(), e =
            listOfLevels[i]->end(); it != e; ++it){
309
310       IdType tmp = *it, tmp_new;
311       if(arrayOfEdges[tmp]->size() > 0){
312         fileWithTmpDecomposition << endl;
313       }
314       bool print = false;
315       if(arrayOfEdges[tmp]->size() > 0){
316         do
317         {
318           print = true;
319           GotoLine(nodeNames, tmp);
320           getline(nodeNames,line);
321           trim(line);
322           (vm.count("file")) ? (fileWithTmpDecomposition << "(" << tmp << ")") :
                (fileWithTmpDecomposition << line);
323           tmp_new = arrayOfEdges[tmp]->at(0);
324           arrayOfEdges[tmp]->clear();
325           tmp = tmp_new;
326         } while (arrayOfEdges[tmp]->size() > 0);
327         if(print && tmp != arrayOfEdges.size()-1){
328           GotoLine(nodeNames, tmp);
329           getline(nodeNames,line);
330           trim(line);
331           (vm.count("file")) ? (fileWithTmpDecomposition << "(" << tmp << ")") :
                (fileWithTmpDecomposition << line);
332         }
333       }
334     }
335
336   }
337   fileWithTmpDecomposition.close();
338
339   for(int i = 0; i < arrayOfEdges.size(); i++) delete arrayOfEdges[i];
340   for(int i = 0; i < listOfLevels.size(); i++) delete listOfLevels[i];
341
342   //erase empty lines printed to decomposition.dat_tmp
343   name = "decomposition.dat_tmp";
344   std::ifstream fileWithTmpDecompositionRead(name.c_str());
345   std::istream& decomposition = fileWithTmpDecompositionRead;
346   name = "decomposition.dat";
347   std::ofstream fileWithDecomposition(name.c_str(), ios::trunc);
348   while(getline(decomposition, line)){
349     if(line.size() > 0){
350       fileWithDecomposition << line << endl;
351       loggerOut(line);
352     }
353   }
354   fileWithTmpDecompositionRead.close();
355   remove("decomposition.dat_tmp");
356
357   fileWithNodeNameRead.close();
358   fileWithDecomposition.close();
359   output->flush();
360   output->close();
361   delete output;
362   remove("nodes.dat");
363   remove("decomposition.dat");
```

```
364 }
365 // helper to load the Inverse poset into the memory
366 void makeBoolPoset(Graph *g, boost::program_options::variables_map vm){
367   Graph graph = *g;
368   int layer;
369   string name = "nodes.dat";
370   std::ofstream fileWithNodeName(name.c_str(), ios::trunc);
371   boost::shared_ptr<Node> root;
372
373   vector<NameElementType> rootName;
374   root = boost::make_shared<Node>(rootName,0,0);
375   root->boolSize = vm["bool"].as<int>();
376   string fileName = boost::lexical_cast<string>(root->boolSize)+".txt";
377   output = new std::ofstream(fileName.c_str(), ios::trunc);
378   arrayOfEdges.push_back(new vector<IdType>());
379
380   name = root->getName();
381   int stringSize = (root->boolSize <= 9 ? 2+root->boolSize*2 : 2+9*2+(root->
          boolSize-9)*3);
382   fileWithNodeName << name;
383   for(int k = 0; k < stringSize-name.length(); k++)fileWithNodeName << "␣";
384   fileWithNodeName << endl;
385
386   map<vector<NameElementType>, boost::shared_ptr<Node> >* layerAct  = new map<
          vector<NameElementType>, boost::shared_ptr<Node> >;
387   layerAct->operator[](root->name) = root;
388
389   vector<IdType>* tmp_level = new vector<IdType>();
390   tmp_level->push_back(root->id);
391   listOfLevels.push_back(tmp_level);
392   map<vector<NameElementType>,  boost::shared_ptr<Node> >* layerNext;
393   map<vector<NameElementType>, boost::shared_ptr<Node> >::const_iterator it, e;
394
395   // generating all elements
396   while(true){
397     layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node> >();
398     tmp_level = new vector<IdType>();
399
400     for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it)
401     {
402
403       it->second->generateNextBool(layerNext, &arrayOfEdges, tmp_level, &
              fileWithNodeName);
404     }
405     if(tmp_level->size() == 0) break;
406     listOfLevels.push_back(tmp_level);
407
408     layerAct->clear();
409     delete layerAct;
410
411     layerAct = layerNext;
412
413   }
414   // include small loop for the top node for printing decomposition
415   arrayOfEdges[arrayOfEdges.size()-1]->push_back(arrayOfEdges.size()-1);
416   delete layerAct;
417   fileWithNodeName.close();
418   // calling Python diagram class to generate pictures
419   if(listOfLevels.size() > 1){
```

```
420     findSSCD(&graph, vm);
421     if(vm.count("no_pict") == 0){
422       Py_Initialize();
423       PyRun_SimpleString("import os,sys");
424       PyRun_SimpleString("sys.path.append(os.getcwd())");
425       PyRun_SimpleString("import diagram");
426       ofstream py("python.in");
427       py << "f = open('" << fileName << "','r')\nf.close()\ndiagram.
              makeGraphOfBool(f,"<< root->boolSize <<")";
428       py.close();
429       FILE *fp = fopen("python.in", "r");
430       PyRun_SimpleFile(fp,"python.in");
431       fclose(fp);
432
433       Py_Finalize();
434       remove("python.in");
435     }
436   }
437   else
438     loggerOut("--The poset has only one vertex. The SSCD is obvious.--");
439 }
440 // helper to load the Inverse poset into the memory
441 void makeInversePoset(Graph *g, boost::program_options::variables_map vm){
442   Graph graph = *g;
443   int layer;
444   string name = "nodes.dat";
445   std::ofstream fileWithNodeName(name.c_str(), ios::trunc);
446   boost::shared_ptr<Node> root;
447
448   vector<NameElementType> rootName(vm["inversion"].as<vector<int> >().begin(),vm[
          "inversion"].as<vector<int> >().end());
449   root = boost::make_shared<Node>(rootName,0,0);
450   string fileName = root->getName().substr(1,root->getName().size()-2)+".txt";
451   output = new std::ofstream(fileName.c_str(), ios::trunc);
452   arrayOfEdges.push_back(new vector<IdType>());
453
454   fileWithNodeName << root->getName()<<endl;
455   map<vector<NameElementType>, boost::shared_ptr<Node> >* layerAct  = new map<
          vector<NameElementType>, boost::shared_ptr<Node> >;
456   layerAct->operator[](root->name) = root;
457
458   vector<IdType>* tmp_level = new vector<IdType>();
459   tmp_level->push_back(root->id);
460   listOfLevels.push_back(tmp_level);
461   map<vector<NameElementType>,  boost::shared_ptr<Node> >* layerNext;
462   map<vector<NameElementType>, boost::shared_ptr<Node> >::const_iterator it, e;
463
464   // generating all elements
465   while(true){
466     layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node> >();
467     tmp_level = new vector<IdType>();
468
469     for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it)
470     {
471       it->second->generateNextInverse(layerNext, &arrayOfEdges, tmp_level, &
              fileWithNodeName);
472     }
473     if(tmp_level->size() == 0) break;
474     listOfLevels.push_back(tmp_level);
```

```
475
476       layerAct->clear();
477       delete layerAct;
478
479       layerAct = layerNext;
480
481     }
482     // include small loop for the top node for printing decomposition
483     arrayOfEdges[arrayOfEdges.size()-1]->push_back(arrayOfEdges.size()-1);
484     delete layerAct;
485     fileWithNodeName.close();
486     // calling Python diagram class to generate pictures
487     if(listOfLevels.size() > 1){
488       findSSCD(&graph, vm);
489       if(vm.count("no_pict") == 0){
490         Py_Initialize();
491         PyRun_SimpleString("import os,sys");
492         PyRun_SimpleString("sys.path.append(os.getcwd())");
493         PyRun_SimpleString("import diagram");
494         ofstream py("python.in");
495         py << "f = open('" << fileName << "','r')\nf.close()\ndiagram.
               makeGraphOfInversePartialyOrderedMultiSet(f,["<< root->getName().substr
               (1,root->getName().size()-2) <<"])";
496         py.close();
497         FILE *fp = fopen("python.in", "r");
498         PyRun_SimpleFile(fp,"python.in");
499         fclose(fp);
500
501         Py_Finalize();
502         remove("python.in");
503       }
504     }
505     else
506       loggerOut("--The poset has only one vertex. The SSCD is obvious.--");
507 }
508 // helper to load the Young's lattice into the memory
509 void makeYoungsLattice(Graph *g, boost::program_options::variables_map vm){
510   Graph graph = *g;
511   int layer;
512   string name = "nodes.dat";
513   std::ofstream fileWithNodeName(name.c_str(), ios::trunc);
514   boost::shared_ptr<Node> root;
515
516   int m = vm["young"].as<vector<int> >()[0], n = vm["young"].as<vector<int> >()
          [1];
517   vector<NameElementType> rootName;
518   for (int i = 0; i < n; i++){
519     rootName.push_back(0);
520   }
521
522   root = boost::make_shared<Node>(rootName,0,0);
523   string fileName = "";
524   for(int i = 1; i < n; i++) fileName += lexical_cast<string>(m) + ",";
525   fileName += lexical_cast<string>(m) + ".txt";
526   output = new std::ofstream(fileName.c_str(), ios::trunc);
527   arrayOfEdges.push_back(new vector<IdType>());
528
529   fileWithNodeName << root->getNameYoung() << endl;
```

```
530   map<vector<NameElementType>, boost::shared_ptr<Node> >* layerAct = new map<
          vector<NameElementType>, boost::shared_ptr<Node> >;
531   layerAct->operator[](root->name) = root;
532
533   vector<IdType>* tmp_level = new vector<IdType>();
534   tmp_level->push_back(root->id);
535   listOfLevels.push_back(tmp_level);
536   map<vector<NameElementType>,  boost::shared_ptr<Node> >* layerNext;
537   map<vector<NameElementType>, boost::shared_ptr<Node> >::const_iterator it, e;
538
539   // generating all elements
540   while(true){
541     layerNext = new map<vector<NameElementType>, boost::shared_ptr<Node> >();
542     tmp_level = new vector<IdType>();
543
544     for (it = layerAct->begin(), e = layerAct->end(); it != e; ++it)
545     {
546       it->second->generateNextYoung(layerNext, &arrayOfEdges, tmp_level, &
              fileWithNodeName,m);
547     }
548     if(tmp_level->size() == 0) break;
549     listOfLevels.push_back(tmp_level);
550
551     layerAct->clear();
552     delete layerAct;
553
554     layerAct = layerNext;
555
556   }
557   // include small loop for the top node for printing decomposition
558   arrayOfEdges[arrayOfEdges.size()-1]->push_back(arrayOfEdges.size()-1);
559   delete layerAct;
560   fileWithNodeName.close();
561   // calling Python diagram class to generate pictures
562   if(listOfLevels.size() > 1){
563     findSSCD(&graph, vm);
564     if(vm.count("no_pict") == 0){
565       Py_Initialize();
566       PyRun_SimpleString("import os,sys");
567       PyRun_SimpleString("sys.path.append(os.getcwd())");
568       PyRun_SimpleString("import diagram");
569       ofstream py("python.in");
570       py << "f = open('" << fileName << "','r')\nf.close()\ndiagram.
              makeGraphOfYoungsLattice(f,"+lexical_cast<string>(m)+",
              "+lexical_cast<string>(n)+")";
571       py.close();
572       FILE *fp = fopen("python.in", "r");
573       PyRun_SimpleFile(fp,"python.in");
574       fclose(fp);
575
576       Py_Finalize();
577       remove("python.in");
578     }
579   }
580   else
581     loggerOut("--The poset has only one vertex. The SSCD is obvious.--");
582 }
583 // helper to load a poset from the given file into the memory
584 void readPosetFromFile(Graph *g, boost::program_options::variables_map vm){
```

```
585   // Load all nodes
586   Graph graph = *g;
587   string fileName = vm["file"].as<string>()+".txt";
588   output = new std::ofstream(fileName.c_str(), ios::trunc);
589   string name = vm["file"].as<string>()+"_nodes";
590   std::ifstream ifFile(name.c_str());
591   std::istream& isFile = ifFile;
592   string line;
593   while (getline(isFile, line)){
594     vector<string> strs;
595     boost::split(strs, line, boost::is_any_of(","));
596     arrayOfEdges.push_back(new vector<IdType>());
597     for(int i = 0; i < strs.size(); i++){
598       if(strs[0].size() == 0) continue;
599       if(strs.at(i).at(strs.at(i).size()-1) == 13) strs.at(i).resize(strs.at(i).
              size() - 1);
600       arrayOfEdges[arrayOfEdges.size()-1]->push_back(lexical_cast<IdType>(strs.at
              (i)));
601     }
602   }
603   ifFile.close();
604
605   // load the level structure
606   name = vm["file"].as<string>()+"_levels";
607   ifFile.open(name.c_str());
608   isFile.copyfmt(ifFile);
609   while (getline(isFile, line)){
610     vector<string> strs;
611     boost::split(strs, line, boost::is_any_of(","));
612     listOfLevels.push_back(new vector<IdType>());
613     for(int i = 0; i < strs.size(); i++){
614       if(strs.at(i).at(strs.at(i).size()-1) == 13) strs.at(i).resize(strs.at(i).
              size() - 1);
615       listOfLevels[listOfLevels.size()-1]->push_back(lexical_cast<IdType>(strs.at
              (i)));
616     }
617   }
618   // calling Python diagram class to generate pictures
619   if(listOfLevels.size() > 1){
620     findSSCD(&graph, vm);
621     if(vm.count("no_pict") == 0){
622       Py_Initialize();
623       PyRun_SimpleString("import os,sys");
624       PyRun_SimpleString("sys.path.append(os.getcwd())");
625       PyRun_SimpleString("import diagram");
626       ofstream py("python.in");
627       py << "f = open('" << fileName << "','r')\nf.close()\ndiagram.
              makeGraphFromFile(f,'"+vm["file"].as<string>()+"')";
628       py.close();
629       FILE *fp = fopen("python.in", "r");
630       PyRun_SimpleFile(fp,"python.in");
631       fclose(fp);
632
633       Py_Finalize();
634       remove("python.in");
635     }
636   }
637   else
638     loggerOut("--The poset has only one vertex. The SSCD is obvious.--");
```

```
639 }
640
641 int main(int argc, char** argv)
642 {
643   time(&start_time);
644   boost::program_options::options_description desc("Allowed options");
645   desc.add_options()
646     ("bool", po::value<int>(), "set the number of elements in a set; --bool 4")
647     ("file", po::value<string>(), "--file [name]\nRead a poset from files. There
          have to be two files in the current folder, '[name]_nodes' and '[name]
          _levels'. Each line of '[name]_nodes' contains a list of neighbors of
          node with id equal to the line number starting from zero. Each line of '[
          name]_levels' contains a list of nodes in the layer which number
          corresponds to the number of the current line starting from zero.\n[name]
          _levels:\n0\n1,2\n3\n\n[name]_nodes:\n1,2\n3\n3\n3")
648     ("inversion", po::value< vector<int> >()->multitoken(), "element with rank 0;
          --inversion 1 2 3 4 [7 ...]")
649     ("young", po::value< vector<int> >()->multitoken(), "set m and n values; --
          young 3 4")
650     //("sum", po::value<int>(), "construct a poset using sum decomposition up to
          given number; --sum_decomposition 5")
651     ("no_pict", "will not generate pictures")
652     ("help", "produce help message")
653   ;
654   po::variables_map vm;
655   po::store(po::parse_command_line(argc, argv, desc), vm);
656   po::notify(vm);
657
658   if (vm.count("help") || ( !vm["young"].empty() && vm["young"].as<vector<int>
          >().size() != 2 ) || ( vm["young"].empty() && vm["inversion"].empty() && vm
          ["sum"].empty() && vm["file"].empty() && vm["bool"].empty() ) ) {
659     cout << desc << "\n";
660     return 1;
661   }
662
663   Graph graph;
664   if (!vm["bool"].empty())
665     makeBoolPoset(&graph,vm);
666   if (!vm["file"].empty())
667     readPosetFromFile(&graph,vm);
668   if (!vm["inversion"].empty())
669     makeInversePoset(&graph,vm);
670   if (!vm["young"].empty())
671     makeYoungsLattice(&graph,vm);
672   return 0;
673 }
```